Compilation Techniques for Reconfigurable Analog Devices

by

Sara Achour

B.S., University of California, Los Angeles (2013)
S.M., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author .....
Department of Electrical Engineering and Computer Science
August 27, 2021

Certified by .....
Martin Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Compilation Techniques for Reconfigurable Analog Devices

by

Sara Achour

Submitted to the Department of Electrical Engineering and Computer Science
on August 27, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Reconfigurable dynamical-system solving analog devices are a powerful new ultra-low-power computing substrate capable of executing dynamical systems in a performant and energy-efficient manner. This class of devices leverages the physical behavior of transistors to directly implement computation. Under this paradigm, voltages and currents within the device implement continuously evolving variables in the computation. These hardware platforms are challenging to use because they are subject to a variety of low-level physical behaviors that profoundly affect the computation. Relevant physical behaviors include operating range and frequency limitations, noise, process variation, and quantization error.

In this thesis, I present compilation techniques for automatically configuring such devices to execute dynamical systems and present the first compiler that automatically targets a physical dynamical system-solving reconfigurable analog device of this class. The presented compiler frees the end user from reasoning about the low-level physical behaviors present in the hardware and automates the process of mapping the dynamical system to the analog hardware. This thesis also introduces specification languages for describing dynamical systems, and the capabilities and physical limitations of the reprogrammable analog hardware. The compiler targets these specifications when mapping the computation.

To faithfully implement a computation, the compiler configures the device so that the original dynamical system dynamics can be recovered from the physics of the device at runtime. The mapped computation simultaneously leverages the device physics to implement the desired computation, respect the physical limitations of the device, and attenuate away the unwanted physical behaviors present in the analog hardware. The compiler configures and composes together the analog blocks and simultaneously accounts for all of the low-level behaviors present in the device.

The compiler first maps the target dynamical system to the analog hardware and then transforms the produced circuit to attenuate away unwanted analog behavior. The compiler employs a multi-stage, algebraic rewrite-based circuit synthesis procedure to map the dynamical system to the analog hardware. This procedure synthesizes analog circuits that effectively use parametric and specialized analog blocks

and leverage physical laws to perform computation.

The compiler automatically transforms the mapped circuit to attenuate away the unwanted analog behaviors present in the circuit. This transformation transforms the signals to respect the operating range and frequency limitations present in the hardware and reduces the effect of analog noise, quantization error, process variation-induced behavioral deviations on the computation. The transformed circuit preserves the original dynamics of the system such that the original dynamical system variable trajectories can be recovered by applying a compiler-derived recovery transform. The compiler formulates the problem of transforming the circuit as a convex optimization problem – this enables the compiler to optimally identify circuit transformations that maximize circuit characteristics such as execution speed and signal quality.

The compiler deploys a cross-cutting program optimization in which the calibration algorithm and compiler work together to reduce the effect of process variation-induced behavioral variations on the overall computation. This thesis presents the concept of a delta model, a hardware abstraction that captures the device-specific behavioral deviations present in the calibrated analog hardware. The compiler uses this hardware abstraction to compensate for behavioral variations for the specific device at hand while transforming the circuit. This optimization involves all parts of the software stack. I introduce delta model language constructs to the hardware specification language, develop a novel delta-model aware circuit scaling optimization, and introduce new calibration and characterization procedures into the device runtime and firmware to implement this optimization. With this optimization enabled, I am able to attain higher fidelity results with more consistency on the target hardware. This thesis also presents a co-designed calibration algorithm that prioritizes eliminating behavioral deviations that cannot be compensated for in compilation.

I evaluate the compiler on applications from the biology, physics, and controls domains. The results demonstrate that these applications execute with acceptable error while consuming microjoules of energy.

Thesis Supervisor: Martin Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I want to thank my advisor, Martin Rinard, for supervising my research and mentoring me over these years. Under his guidance, I felt that I grew as both a researcher and a communicator. I greatly appreciated all of his insights over the years – research-related, philosophical, and otherwise.

I would also like to express my gratitude to Michael Carbin and Yannis Tsividis for serving as readers for this thesis. I greatly appreciated all of the valuable insights they brought to this work. I would like to thank Yannis Tsividis especially for working with me on this project and providing the hardware prototypes I targeted in this research. His expertise and insights were invaluable for this line of work.

I would like to take the time to acknowledge all of my wonderful colleagues from the programming languages and systems communities for all of the insights they've offered over the years. I'd like to thank Sasa Misailovic, Michael Carbin, Stelios Sidiroglou-Douskos, Jeff Perkins, and Michael Gordon for showing me the ropes when I first started as a graduate student and involving me in the various grants we've had throughout the years. I'd like to more generally thank both the current and former members of the PAC group. The camaraderie within the group undoubtedly helped make the most strenuous of deadlines more bearable.

I will always be grateful for all of my mentors throughout undergrad that set me on this path. I'd like to thank Jens Palsberg and Glenn Reinman for introducing me to computer science research. Without their guidance, I would likely have not discovered this career path. I am also forever indebted to Joe DiStefano, who taught mentored me through my undergraduate years. Had I not been a part of the computational systems biology interdepartmental program he supervised, I would not have discovered computer science as a field of study.

While this journey was long, it was far from lonely. I will forever be grateful to all of my friends from MIT for being there for me throughout this process. I will fondly remember all of the thanksgivings, patio beers, Dungeons and Dragons sessions, adrenaline-inducing boating excursions that we had over the years. It was

these adventures that really made my time here at MIT special. I'd like to thank my flatmates Eva Golos, Deborah Pohlmann, Lindsay Brownell, and Kasturi Shah for living with me over the years. Our morning coffees, spontaneous brunches, and post-work outings made even the bleakest of days enjoyable. I would like to also extend my gratitude to all of the people at WMBR for hosting me and giving 1001 a home. Monday nights will always hold a special place in my heart. Finally, I would like to thank my family for ensuring I received a good education and encouraging me to pursue a career in STEM. Without their support, this journey surely would have been more difficult.

This thesis is dedicated to my grandmother, Zaina, who ensured her children received the education that she was denied as a girl. In loving memory of my grandfather, Said.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Specialized computing platforms, implemented on a range of devices including analog, photonic, and digital devices, are becoming pervasive and crucial for satisfying the computational needs of different domains. Examples include specialized devices that efficiently solve problems in machine learning, quantum computing, signal processing, robotics, and biology [120, 56, 54, 11, 108, 85, 105, 29, 128, 31, 140, 132, 51, 61, 141, 15, 109, 102, 62, 87, 89, 38]. Delivering the potential of such devices to domain specialists is a challenge as there is sometimes a significant gap between the programming interface that the device provides and a programming model the end user can use productively. This gap between an effective high-level programming model and the hardware programming interface often occurs in designs that prioritize accuracy, device area, or performance over usability.

I present a new compiler for ultra-low power reconfigurable analog computing platforms which solve dynamical systems [61, 51, 128, 140]. These devices are programmed by routing together configurable analog blocks using digitally programmable interconnects. The programmed computation is then executed by powering on the analog device and observing the voltage and current trajectories over time. These signal trajectories capture the evolution of dynamical system quantities over time. The presented compiler automates the programming process so that these devices are more accessible to programmers. The compiler automatically performs all of the device configuration steps for the end user and automatically reasons about any low-

level physical behaviors present in the device. With this compiler, the end user needs only to specify the dynamical system, variables of interest, and value ranges for each variable. The compiler produces, as output, a configuration for the analog device that encodes a circuit comprised of configured analog blocks. The original dynamical system dynamics can be recovered from the circuit dynamics at runtime by applying a compiler-derived recovery transform.

To faithfully implement the computation, the compiler transforms the computation to reduce the effect of a variety of low-level physical behaviors on the overall computation. Relevant low-level physical behaviors include operating range and frequency limitations, noise, process variation-induced behavioral deviations, and quantization error. These low-level behaviors can have a profound effect on the fidelity of the mapped computation. The compiler transforms the mapped computation to respect the physical limitations of the device and attenuate away the unwanted physical behaviors present in the analog blocks – this reduces the effect of these low-level physical behaviors on the computation. The original dynamical system dynamics can be recovered from the transformed computation at runtime by applying a compiler-derived recovery transform. The compiler, therefore, frees the end user from reasoning about and compensating for the low-level physical behaviors present in the hardware when designing the computation.

The compiler deploys a cross-cutting program optimization in which the device calibration algorithm and compiler work together to reduce the effect of process variation-induced behavioral variations on the overall computation. Each block on the analog device is designed to implement some function g. However, due to the effects of process variation, the blocks rarely implement g accurately post-fabrication. To mitigate this issue, designers introduce calibration circuits into the hardware. With this addition, each block in the analog hardware can be calibrated to implement a range of functions $f_1 ... f_n$.

Traditional Approach: Traditionally, the compiler targets the function g that each block is designed to implement. The calibration algorithm then calibrates each block to implement the function f_i that most closely approximates the function g. With this

traditional approach, the compiler may produce circuits that inaccurately execute the target computation. These inaccuracies occur when the circuit includes blocks that cannot be calibrated to closely approximate the function g.

Co-Designed Approach: This work deploys a cross-cutting compiler optimization which enables the compiler to target a range of functions $g_1 ... g_m$ for each block. Under this paradigm, the calibration algorithm identifies the function pair $\langle f_i, g_j \rangle$ in which f_i most closely approximates g_j over all pairs of f and g functions. Because each block on the device may be calibrated to implement a range of functions, the calibration algorithm is free to select a function g_j for each block that can be implemented with the smallest error in hardware. This approach introduces less error into the computation because each block in the associated circuit may be calibrated to implement the function that delivers the lowest error.

This thesis introduces a delta model specification construct that codifies the space of functions $g_1 ... g_m$ for each block and a delta model hardware abstraction that describes, for each block instance in the device on hand, the function g_j the block instance has been calibrated to most closely approximate. The compiler uses the delta model hardware abstraction to compensate for behavioral variations present in the calibrated hardware when transforming the circuit. This compiler optimization enables the compiler to reduce the effect of process variation on the overall computation. I also present a co-designed calibration algorithm that selects the function g_j the target block can be calibrated to most closely approximate and calibrates the block to implement the desired function.

The compiler automatically maps the target dynamical system to the analog hardware. The compiler productively configures and composes together programmable mixed-signal blocks that implement a host of non-standard, complex functions and leverages physical laws, such as Kirchhoff's law, to implement the desired computation. The compiler also introduces special-purpose blocks to convert, copy, and route signals as necessary to implement the desired computation. These capabilities together enable the compiler to effectively map the target dynamical system to the analog hardware.

The compiler presented in this thesis works with a specification of the target analog device written in the analog device specification language. The analog device specification language offers language constructs for describing the programmable blocks, programmable connections, and the low-level physical behaviors present in the target device. Because the compiler targets a specification of the analog device, it is capable of targeting a range of devices. Specific devices include simulated devices based on mixed-signal gene-protein network accelerators [128, 140] and the HCDCv2 differential equation solving analog device[61, 51, 132, 61, 51]. I evaluate the compiler on the HCDCv2 differential-equation solving analog device [61, 51, 132]. This compiler is the first to target any simulated or fabricated reconfigurable differential equation-solving analog device. I evaluate the compiler on a broad range of dynamical systems from the physics, biology, and controls domains. The automatically generated configurations execute the described dynamical system computations with acceptable error while consuming significantly less energy than corresponding digital computations.

1.1 Dynamical Systems

A dynamical system is a system whose state evolves over time. Dynamical systems appear in a wide variety of fields including mathematics, physics, chemistry, biology, economics, engineering, machine learning, signal processing, and medicine [74, 117, 78, 14, 33, 9, 20, 143, 19, 45, 12, 25, 47, 118]. In the biological and physical sciences, researchers use dynamical systems to predict and understand the behavior of physical processes [74, 117]. For example, medical practitioners may use a dynamical system to model the effect of an injection on an individual's hormone levels and then use this information to derive an initial dosage.

Dynamical systems also often appear in systems which interact with the environment. Dynamical systems can be used to reconstruct higher-order information from environmental signals and control actuators (e.g. motors) in real-time [78, 14, 33, 9]. For example, a drone may use a dynamical system to adjust the speed of its rotors based on wind conditions.

Dynamical systems are typically implemented with ordinary differential equations (ODEs) or partial differential equations (PDEs). In this work, I focus on dynamical systems comprised of ordinary differential equations. An ordinary differential equation (ODE) is a differential equation in which all derivatives are taken with respect to an independent variable, typically time. Practitioners generally are interested in *simulating* a dynamical system. Dynamical system simulators and solvers compute the trajectories of the state variables over simulation time.

While dynamical systems are invaluable in many different domains, there are challenges with simulating dynamical systems accurately and performantly on digital hardware. Digital ODE solvers have difficulty efficiently simulating non-linear differential equations or differential equations with dynamics which operate on different time scales. To mitigate these issues, practitioners linearize the differential equations or replace fast-evolving differential equations with closed-form solutions [107, 41, 133, 58, 98, 99, 17]. These optimizations approximate the dynamical system but enable the digital solver to more efficiently simulate the system.

These efficiency issues are especially important for dynamical system applications with real-time performance requirements. Real-time dynamical systems typically interface with the environment and appear in robotics, communications, and feature recognition (e.g. voice recognition) applications [129, 106, 139]. To function correctly, the real-time dynamical system must continuously process sensor inputs or continuously tune actuator parameters without falling behind. Because these applications are typically run on embedded systems, these dynamical system implementations must meet real-time performance requirements on energy- and compute-constrained platforms. Typically, practitioners discretize or aggressively simplify real-time dynamical systems to improve performance [139]. Even with these simplifications, performantly executing such computations in energy- and compute-constrained environments remains a challenge.

1.2 Analog Computing

Historically, practitioners simulated dynamical systems with analog computers. This form of analog computing primarily involved manually building analog circuits from basic electrical components such as resistors, capacitors, and inductors to simulate dynamical systems [130, 34, 104, 91]. These historical analog computers offered high-precision analog primitives that could be manufactured at low tolerances. These functional units performed computation at high accuracy – usually within 0.01% error relative to the full-scale range of the signal for linear components [65].

Typically, the programmer would draft a circuit whose physical behavior was analogous to the dynamical system dynamics. In these circuits, the physics of the voltages in the circuit over time match the dynamics of the target dynamical system over time. To execute the dynamical system, the practitioner would power on the analog circuit and observe the trajectories of the analog voltages of interest for a period of time. If the practitioner designed the circuit correctly, the observed current and voltage trajectories would follow the same path as the dynamical system variable trajectories. Researchers favored performing numerical computations with analog hardware because digital hardware was not yet mature enough to perform these computations efficiently. These historical analog computing platforms disappeared as the performance of digital computers improved.

1.2.1 Modern Analog Computing

In recent years, analog computation has been experiencing a renaissance in the hardware community. Hardware designers have put forth a variety of modern day digitally programmable electrical analog devices [51, 61, 54, 11, 105, 29, 128, 31, 140, 109, 15, 102]. These analog devices use standard CMOS processes and leverage transistor physics to perform computation. This line of research focuses on ultra-low power reconfigurable electrical analog devices which simulate dynamical system computations [61, 51, 128, 140].

This class of modern dynamical system-solving analog devices leverages the ana-

log behavior of transistors to implement computation. Under this paradigm, voltages and currents within the device capture the dynamics of the continuously evolving variables in the dynamical system. This computational model closely resembles the computational model employed by historical analog devices. Because modern devices make use of standard CMOS processes, they are much smaller than their historical counterparts. These devices also offer digitally programmable interconnects and values and are much easier to automatically configure and integrate with digital systems.

Unlike historical analog computers, modern dynamical system-solving analog devices are engineered for energy efficiency and capable of performing computation with very little power [51, 61]. These modern analog devices provide low-power approximate computational blocks that incur between 1%-5% error. These low-precision blocks are difficult to manufacture at low tolerances and are therefore sensitive to the effects of process variation [132, 51, 61]. These modern analog blocks are also subject to operating range and frequency limitations and are sensitive to noise. In contrast, historical analog devices consumed much more power and area but provided high precision components that could be manufactured at low tolerances and were therefore less affected by process variation. Because modern analog devices are inherently approximate devices, they require the development of fundamentally new programming techniques. Applying the programming techniques used for historical analog computers would produce unacceptably inaccurate results on this class of modern hardware.

Benefits of Modern Dynamical System-Solving Analog Devices

Dynamical system-solving analog devices are attractive computational targets because they are low power devices and have predictable performance characteristics [51, 61]. In these systems, the mapping between dynamical system simulation time and wall-clock time can be statically computed with high accuracy at compile-time. This mapping defines how many milliseconds of wall-clock time it takes to simulate the dynamical system for one unit of simulation time. Practitioners can then use this mapping to determine if the analog implementation of the dynamical

system meets the performance requirements of the target computation.

Today, dynamical systems are simulated with digital dynamical system simulators. Digital simulators do not typically have predictable performance characteristics when simulating non-linear and stiff dynamical systems [107, 41, 133, 58, 98, 99, 17]. Statically inferring a tight execution time bound for digital dynamical system simulators therefore remains a challenge.

This class of modern analog devices is also capable of efficiently simulating complex dynamical systems. Because dynamical system-solving analog devices are highly parallel spatial computing substrates, the execution time of the dynamical system does not necessarily increase with dynamical system size and complexity.

These performance characteristics together enable practitioners to execute dynamical systems which execute predictably and performantly while consuming very little energy. In addition, some dynamical system-solving analog devices can compute on externally provided analog signals and support emitting analog signals to externally accessible hardware interfaces [51, 61]. These devices therefore support low power, realtime signal processing applications that work with analog sensors and actuators.

Programming Modern Dynamical System-Solving Analog Devices

Modern dynamical system-solving analog devices are programmed by routing together configurable analog blocks using programmable interconnects to form an analog circuit. For the programmed circuit to faithfully implement the dynamical system, the original dynamical system variable trajectories must be recoverable from the voltage and current trajectories at runtime. The original dynamical system trajectories are recovered by applying a recovery transform to the measured signal trajectories. The recovery transform maps signal values to dynamical system variable values and wall-clock time samples to simulation time samples. A dynamical system is recoverable from a programmed circuit if it can be recovered at runtime through the use of a recovery transform.

Physical Limitations, Noise, and Quantization Error: This class of reconfigurable analog devices is subject to a variety of low-level physical effects which affect

the fidelity of the computation. Analog blocks impose frequency, current, and voltage range limitations and have unique noise characteristics. The digital interfaces to analog blocks are subject to the effects of quantization error and encode a limited range of digital values. The practitioner must account for all of these low-level physical behaviors when programming the hardware.

Process Variation: Analog blocks are also subject to process variation-induced variations in behavior – these unwanted behaviors appear post-fabrication and vary from device to device. Hardware designers typically introduce calibration circuits into their design to reduce the effect of process variation on the overall computation. These calibration circuits are tuned online post-fabrication to eliminate unwanted behavior from the device on hand. These calibration circuits either self-tune autonomously or are configured by an algorithm implemented in the device firmware. In some cases, the calibration circuits fail to completely eliminate the process variation-induced behavioral variations from the analog blocks. This causes the calibrated block's behavior to deviate from its expected behavior and introduces error into the computation. In extreme cases, practitioners may need to hand-select well-behaved blocks or manually account for behavioral deviations present in the device on-hand when programming the hardware.

Prior to the techniques presented in this thesis, practitioners had to manually program these state-of-the-art dynamical system-solving analog devices to implement the desired computation. This process involves manually drafting an analog circuit composed of configured blocks. Because these blocks are implemented with analog circuits, practitioners must also account for a host of low-level physical behaviors, including noise, process variation, and signal and frequency range limitations. To do so, practitioners must manually transform the programmed circuit to account for these behaviors while simultaneously ensuring the original dynamical system is recoverable at runtime. This arduous, error-prone programming process remains a significant barrier to adopting these platforms today.

1.3 Problem Statement

My thesis research presents a compiler for reconfigurable dynamical system-solving analog devices. The compiler automatically generates an analog circuit composed of configured blocks and transforms the circuit to account for the low-level physical behaviors present in the hardware. The compiler accepts, as input, a specification of the dynamical system and a specification of the analog device. The compiler produces, as output, an analog device program (ADP) which can be executed on the analog hardware. The analog device program configures and connects together a subset of blocks resident on the target analog device and specifies the recovery transform. The recover transform recovers the original dynamical system dynamics from the ADP signal trajectories. The end user provides the dynamical system specification, and the hardware designer provides the analog device specification.

The analog device program produced by the compiler implements a circuit whose physical behavior captures the behavior of the target dynamical system such that the original dynamical system is recoverable at runtime. The compiler automatically derives a recovery transform that recovers the original dynamical system as part of the compilation process. To our knowledge, this is the first compiler to target a simulated or fabricated differential-equation solving reconfigurable analog device of this class.

1.3.1 Challenges

Reconfigurable analog computing platforms present fundamentally different programming challenges than digital computing platforms:

- *The low-level physics of the device can have a fundamental effect on the computation* – analog blocks impose frequency, current, and voltage range limitations and have unique noise and error characteristics. Analog blocks are also subject to process variation-induced variations in behavior. All of these low-level physical behaviors must be taken into consideration when programming the device. This is especially true for modern incarnations of analog devices which

often offer low precision, approximate blocks and are sensitive to the effects of process variation. Failing to adequately consider these low-level behaviors introduces more error into an already approximate computation – this may cause the mapped program to execute with unacceptable error.

- *The provided analog blocks may implement complex functions* – analog devices provide highly specialized blocks that implement anything from simple functions to sets of differential equations. These blocks are highly configurable and often can be reconfigured to implement a multitude of different functions. Because there is no universally agreed upon collection of analog blocks, the set of provided programmable blocks may vary across analog computing platforms. Furthermore, two blocks of the same type may not implement the same set of functions in practice due to variations introduced during fabrication.

- *Analog blocks cannot be arbitrarily routed together* — the programmable interconnects are heavily constrained and limited in quantity. Therefore, computations must be carefully laid out on the device so that all the necessary connections can be made.

All of these factors together make reconfigurable analog devices a challenging compilation target that requires fundamentally new techniques.

1.3.2 Advancement over State of the Art

This thesis introduces new languages which together define the compilation problem. The compiler maps programs written in the dynamical system specification language to an analog device. The compiler works with a specification of the device, written in the analog device programming language. The compiler produces, as output, a transformed analog circuit, written in the analog device programming language:

- **Dynamical System Specification Languages** - In this thesis, I introduce a specification language for defining dynamical systems. The dynamical system specification language (DSSL) provides constructs for specifying systems of

first-order differential equations. The DSSL is the high-level language targeted by the compiler.

- **Analog Device Specification Language**: This thesis presents a novel specification language for reconfigurable analog devices. The analog device specification language (ADSL) offers block specification constructs for defining the programming interface, dynamics, physical behaviors, and physical limitations of each block. The specification language also provides constructs for specifying the spatial layout of blocks on the device and the available digitally programmable connections offered by the device. A key challenge with designing the ADSL is identifying the right abstractions for the programming interface and low-level physical behaviors.

- **Analog Device Programming Language**: This thesis also presents a programming language (ADPL) for analog devices. The analog device programming language offers constructs for configuring blocks and enabling digitally programmable connections within the device. These two constructs together are used to describe a circuit of configured blocks on the device. The ADPL also supports defining a recovery transform – this recovery transform recovers the original dynamical system dynamics at runtime.

The compiler presented in this thesis leverages the following compilation techniques to target the analog device effectively:

- **Circuit Synthesis** - This thesis presents a novel circuit synthesis procedure that derives an analog device program that implements the given dynamical system, subject to the resource and connectivity constraints of the analog device. This circuit synthesis procedure can identify non-trivial usages of the available blocks to implement the desired circuit and use specialized blocks when necessary to forward, convert, and copy signals.

- **Circuit Scaling** - This thesis presents a novel circuit scaling procedure that automatically transforms an input analog device program to abide by the physical

restrictions imposed by the device. The resulting transformed circuit captures the original dynamical system such that the original dynamical system is recoverable at runtime. The circuit scaling procedure derives a transform that automatically compensates for process variation-induced behavioral deviations present in the device. This compensation operation is critical for obtaining accurate executions on modern analog devices.

- **Synergistic Calibration and Compilation** - This thesis presents a novel cross-cutting compiler optimization in which the device calibration routines and compiler work together to reduce the effect of process variation-induced behavioral deviations on the overall computation.

  Traditionally, architects design calibration algorithms to eliminate the subset of the behavioral deviations that can be attenuated away with the calibration circuitry. I introduce a co-designed calibration algorithm that prioritizes eliminating behavioral variations that the compiler cannot handle during compilation.

  I introduce the concept of a delta model, a new hardware abstraction that captures behavioral deviations present in the calibrated hardware. The compiler targets the delta model representation of the behavioral deviations when targeting the device on hand. The compiler identifies and compensates for any correctable behavioral deviations when scaling the circuit. This abstraction provides the compiler with a more accurate representation of the empirically observed behavior of the calibrated device on hand. The compiler uses these models to produce programs that deliver better end-to-end accuracy. All uncorrectable behavioral deviations are accepted as part of the behavior of the block and introduce error into the overall computation.

  This optimization involves all parts of the software stack. I introduce delta model language constructs to the ADSL, develop a novel delta-model aware circuit scaling optimization, and introduce new calibration and characterization procedures into the device runtime and firmware to implement this optimization.

With this optimization enabled, I am able to attain higher fidelity results with more consistency on the target hardware.

1.3.3 Circuit Synthesis

Because analog blocks are often engineered for efficiency and generality rather than ease of use, there may be a substantial semantic gap between the dynamical system and the analog hardware. For example, some analog devices may only provide computational blocks that implement complicated functions which are difficult to compose together.

The goal of the circuit synthesis procedure is to derive an analog device program that implements the given dynamical system, subject to the resource and connectivity constraints of the analog device. The circuit synthesis procedure ensures the physics of the circuit implemented by the derived analog device program is algebraically equivalent to the dynamics of the dynamical system. Two relations are algebraically equivalent if they produce the same output over all possible inputs. The circuit synthesis procedure works with an idealized representation of the hardware which is not subject to low-level physical behaviors.

The circuit synthesis procedure automatically derives an analog circuit comprised of configured analog blocks that implements the target dynamical system. The circuit synthesis algorithm can identify non-trivial usages of the available blocks to implement the desired circuit. The compiler also employs a spatially aware routing procedure that can successfully map circuits in the presence of a restrictive routing environment. The circuit synthesis procedure produces an analog device program that encodes a circuit which is algebraically equivalent to the starting dynamical system. In this thesis, I refer to an ADP produced by the circuit synthesis procedure as an unscaled ADP.

1.3.4 Circuit Scaling

The physical behavior of an analog device has a profound impact on the implemented computation. Relevant phenomena include quantization error, noise, manufacturing variations, and frequency, current, and voltage range limitations. For an analog device program to faithfully implement a dynamical system, it must not violate the physical constraints of the analog hardware. A compilation goal is therefore to transform the analog device program to abide by the physical restrictions imposed by the device, while ensuring the dynamics of the original dynamical system can be recovered at runtime.

The circuit scaling procedure computes a scaling transform comprised of magnitude scale factors which scale the values and signals in the ADP and a time scale factor which changes the execution speed of the computation. The circuit scaling procedure applies the scaling transform to the provided unscaled ADP to produce a scaled ADP. The scaling transform is applied to the unscaled ADP by multiplying each digitally settable value by its magnitude scale factor. This internally sets the execution speed of the computation and scales all the signals in the ADP by their respective magnitude scale factors. The circuit scaling procedure exploits a property of dynamical systems to tune the execution speed of the dynamical system (Section 3.1.2). The scaled ADP specifies a recovery transform that recovers the original dynamical system dynamics at runtime. The specified recovery transform multiplies the signal samples and time samples by statically derived constant factors.

The circuit scaling procedure produces a scaled ADP which respects all of the frequency, current, and voltage range limitations imposed by the device and compensates for the subset of process variation-induced behavioral variations which are amenable to static compensation. The scaled ADP also increases the dynamic range of the signals when possible to reduce the effect of noise and error on the computation. Applying the recovery transform specified in the scaled ADP at runtime recovers the original dynamical system dynamics from the observed signal trajectories.

The circuit scaling procedure frames the core problem of finding a scaling trans-

form as a geometric programming problem. A geometric programming problem is a type of constrained optimization problem that can be solved optimally and efficiently with a numerical solver [18, 92]. The geometric programming problem contains an objective function that encodes the optimality criteria of the optimization problem. Because geometric programming problems can support non-linear constraints, the circuit scaling algorithm can propagate the scaling transform through non-linear analog blocks. The problem constraints ensure the dynamical system dynamics are recoverable from the scaled ADP and encode the physical limitations of the hardware. The objective function encodes what property of the scaled ADP to optimize. In this thesis, I use an objective function that jointly maximizes the computation speed and the signal-to-noise ratio of the signals and values.

The circuit scaling algorithm presented in this thesis also deploys a delta model compensation optimization which compensates for manufacturing variation-induced behavioral deviations present in the calibrated device. This optimization enables the compiler to tailor the scaling procedure to more effectively target the device on hand. The delta model compensation optimization intelligently scales the circuit to reduce the effect of a subset of correctable behavioral deviations. Here, a correctable behavioral deviation is a deviation which scales a signal or value within an analog block. This optimization augments the geometric programming problem recoverability constraints to incorporate the effect of these correctable deviations on the scaled signals. The delta model compensation optimization works with a set of empirically elicited delta models for the device on hand.

Note that the circuit scaling algorithm presented in this thesis supports partially reprogramming the ADP blocks to better scale the circuit. The algorithm formulates the circuit scaling+block reprogramming problem as a combinatorial geometric programming problem (CGP) which contains both geometric programming constraints and discrete constraints. The circuit scaling algorithm solves the CGP to obtain a set of block reprogramming operations. Once the compiler has reprogrammed the blocks, the CGP simplifies to a geometric programming problem which computes the optimal scaling transform for the ADP.

Evaluation

Using the above techniques, I designed a compilation toolchain that targets the HCDCv2 analog device[51, 61]. The HCDCv2 analog device is an ultra low power reconfigurable analog computing platform designed for running general non-linear dynamical systems. I compile twelve benchmark applications from the biology, physics, and controls domains to the HCDCv2 with the implemented compilation toolchain. The compilation toolchain produces multiple scaled ADPs that all implement the target dynamical system. I execute the produced scaled ADPs on the analog hardware and compare the recovered variable trajectories with reference trajectories computed with a high-precision digital differential equation solver. For all benchmark applications, the compiler is able to identify analog programs that execute the dynamical system on the analog hardware at high fidelity.

I then investigate the effect the presented compiler optimizations have on the fidelity of the end-to-end results. The scaling transform is integral to producing circuits that can be executed on the analog device. The behavioral deviation compensation and block reprogramming optimizations are critical to producing circuits that accurately execute the target dynamical system on the analog hardware. These findings demonstrate that the circuit scaling procedure and its associated optimizations are critical parts of the compilation process.

To the best of my knowledge, this compiler is the first to target any modern programmable analog device for dynamical systems. This thesis is the first to present experimental results for any compiled computation executing on any physical programmable analog device of this class.

1.4 Overview of Thesis

I next present a summary of the topics covered in this thesis. Each section in this summary corresponds to a chapter in the thesis.

1.4.1 Background and Related Work (Chapter 2)

I present an overview of the related work. I first provide an overview of use cases for dynamical systems, outline the types of dynamical systems, and provide an overview of digital simulation approaches for dynamical systems. I then discuss classes of dynamical systems which are difficult to simulate accurately and efficiently with digital solvers. I then provide an overview of the history of analog computing and describe how practitioners programmed historical analog devices in the past. I then present an overview of the kinds of modern reconfigurable analog devices seen today and contrast the software techniques employed by these devices. I conclude this chapter with a discussion of related compilation techniques and numerical methods.

1.4.2 Dynamical Systems (Chapter 3)

I present the high-level dynamical system specification language (DSSL) targeted by the compiler. The dynamical system specification language is a high-level programming language used for describing dynamical systems. I first present an overview of dynamical systems and ordinary differential equations (ODEs) and discuss digital and analog simulation approaches for ODEs. I introduce the time scaling property of ODEs – the compiler leverages this time scaling property to change the execution speed of the computation. I then formally introduce the dynamical system specification language and summarize all of the language constructs.

1.4.3 Dynamical System Applications (Chapter 4)

I present the dynamical system specifications for twelve benchmark dynamical system applications from the biological, physics, and controls domains. I then present the dynamical system specifications for two real-time signal processing applications that continuously perform computation on external analog signals.

1.4.4 Reconfigurable Analog Devices (Chapter 5)

I provide a comprehensive overview of differential equation-solving analog devices and introduce the analog device specification language. In this chapter, I also present the analog device specification for the HCDCv2 hardware platform and provide an overview of the HCDCv2 runtime system and firmware.

Programming Challenges from the Gates Up

I first provide an overview of the high-level and low-level programming interfaces for the target class of devices. I next present an overview of the low-level physical behaviors (unexpected signal biases and gains, frequency-dependent behavior, and noise) present in this class of devices. I then discuss how these behaviors are mitigated (or propagated to higher levels of abstraction) in the device firmware, in the device runtime, in the analog device specification. Common mitigation strategies include calibrating the hardware and imposing physical restrictions such as operating range and frequency limitations on the device. The compiler automatically reasons about the low-level behaviors which are not mitigated at lower levels of abstraction.

Delta Models and Calibration

I next introduce the concept of a delta model – a hardware abstraction that captures process variation-induced behavioral variations present in the calibrated device on hand. This compiler uses this hardware abstraction to produce programs that execute more accurately on the analog device. The calibration algorithm deployed in the device firmware impacts the delta models' ability to capture the behavioral variations present in the calibrated hardware. I refer to the calibration algorithm as the calibration strategy in this thesis. I introduce a traditional calibration strategy which is typically used in hardware design, and a co-designed calibration strategy which is designed with the capabilities of delta models in mind. I then describe the effect these calibration strategies have on the delta model.

Analog Device Specification and Programming Language

I rigorously describe the analog device specification language. The analog device specification language enables hardware designers to define all of the available digitally settable connections and programmable blocks within the device. The language will support the specification of blocks with specialized programming interfaces and provide constructs for describing the spatial layout of these blocks in hardware. The analog device specification language offers language constructs for defining the input-output relation implemented by each block and noise, operating range, and frequency annotations for specifying the physical limitations and low-level physical behaviors present within the block. The analog device specification language also offers constructs for defining delta model specifications. The compiler combines each block's delta model specification with empirically derived delta model information to identify a block's delta models.

I rigorously describe the analog device programming language. The analog device programming language (ADPL) supports the specification of circuits comprised of configured analog blocks. The language offers constructs for programming block values and connecting block ports together with digitally programmable signals. The analog device programming language supports annotating signals within the circuit with dynamical system variables and expressions. The compiler uses these annotations to relate currents and voltages to quantities in the target dynamical system. The programming language also supports the specification of a scaling transform for the described circuit. This transform is applied to the data field values before execution – the resulting scaled circuit respects all of the physical behaviors and limitations imposed on the computation by the analog device. The scaling transform is also used to recover the original dynamical system dynamics from the signal trajectories.

The HCDCv2

I present the analog device specification for the HCDCv2 and describe the calibration procedures and runtime system deployed by the HCDCv2. I first present the

block specifications and device layout specification for the HCDCv2. I then provide an overview of the calibration strategies deployed by the HCDCv2. The calibration strategy dictates how the firmware calibrates the blocks in the HCDCv2. Each calibration strategy prioritizes eliminating a subset of unwanted behaviors within a calibrated block. The HCDCv2 firmware offers two calibration strategies: a traditional calibration strategy (`minimize_error`) and a co-designed calibration strategy (`maximize_fit`). The traditional calibration strategy seeks to calibrate each block to deliver the block input-output relation described in the HCDCv2 analog device specification. The co-designed calibration strategy prioritizes eliminating behaviors that the compiler cannot statically compensate for with a delta model. I provide a detailed multiplier case study which demonstrates how the delta model and calibration strategy interact on a fabricated instance of the HCDCv2.

I then provide an overview of the HCDCv2 runtime and low-level programming interface. I discuss the device calibration and characterization procedures employed by the device runtime and describe how the runtime empirically derives delta model information from the HCDCv2. The HCDCv2 runtime system stores the collected calibration, characterization, and delta model information in calibration, profiling, and delta model databases. I conclude the chapter with an overview of how the HCDCv2 runtime executes an ADP to the HCDCv2.

1.4.5 Scaled and Unscaled ADPs (Chapter 6)

I next present a detailed overview of the analog device programs produced by the compiler. This chapter introduces the concept of an unscaled ADP and a scaled ADP:

- **Unscaled ADPs**: The compiler first produces an unscaled ADP which implements the target dynamical system. In an unscaled ADP, the physics of the currents and voltages are semantically equivalent to the dynamical system dynamics. The unscaled ADP does not consider any of the operating range and frequency limitations of the device or take into account the effect of analog

noise, quantization, or process variation.

- **Scaled ADPs**: The compiler produces a scaled ADP from the unscaled ADP which takes into account all of the physical restrictions and behaviors described above. The scaled ADP specifies a scaling transform that scales all of the programmable values and signals in the circuit so that the circuit dynamics respect the physical constraints of the device.

  A scaling transform is a collection of constant coefficients that describe how all the signals in the program are scaled. The scaling transform is applied at compile-time by multiplying all digitally set values by their respective coefficients. The resulting scaled computation preserves the original behavior of the dynamical system such that it can be recovered at runtime by multiplying the signals by constant values. I discuss this dynamical system preservation property in detail.

I present a cosine ADP case study in which the cosine dynamical system (Section 4.1) is programmed to the HCDCv2 analog device. This case study presents the unscaled and scaled ADPs for the cosine application. I then present the unscaled and scaled ADPs for the twelve benchmark applications introduced in Chapter 4. All of these ADPs were produced by the compiler.

1.4.6 Compilation Overview (Chapter 7)

I provide a high-level overview of the compilation process. In this chapter, I demonstrate how the compiler maps a harmonic oscillator dynamical system computation to a simple reconfigurable analog device (the SIMPL analog device). Each step of compilation is described at a high level and then demonstrated on this running example.

I first introduce the harmonic oscillator dynamical system specification, the SIMPL analog device specification and the unscaled and scaled ADPs for the harmonic oscillator. The dynamical system and analog device specifications are the inputs to the

compiler, and the unscaled and scaled ADPs are the intermediate and final outputs of the compiler.

I then describe the operation of the compiler at a high level. The compiler operates in two phases:

- **Circuit Synthesis (`LGraph`)**: The compiler first synthesizes a circuit comprised of configured blocks that implements the provided dynamical system. The `LGraph` pass first synthesizes a circuit fragment that implements each relation in the dynamical system. It then assembles all of the circuit fragments to form a full circuit that implements the dynamical system. Finally, it maps the circuit blocks to locations on the device and connections to sequences of digitally settable connections in the device. The `LGraph` compilation pass returns an unscaled ADP as output.

- **Circuit Scaling (`LScale`)**: The compiler then derives a scaling transform for the circuit which preserves the integrity of mapped analog computation in the presence of low-level physical behaviors (automated circuit scaling). The `LGraph` pass returns a scaled ADP which implements the dynamical system computation.

1.4.7 Circuit Synthesis (Chapter 8)

I provide a rigorous description of the circuit synthesis pass. This circuit synthesis pass produces analog device programs which are guaranteed to be algebraically equivalent to the provided dynamical system. That is, the analog device program specifies a circuit that implements a dynamical system that can be transformed into the original dynamical system by successively applying algebraic rewrite rules.

The proposed compiler uses a staged circuit synthesis algorithm that efficiently synthesizes circuits by breaking up the compilation process into multiple, more specialized passes. Each pass refines the circuit by adding and configuring blocks. This multi-stage compilation approach enables the compiler to explicitly handle special-use blocks, such as blocks that route and copy signals.

I formally describe the circuit synthesis procedure. I first introduce all of the necessary notation and mathematical constructs used in this chapter. I then rigorously describe each step of circuit synthesis:

- **Circuit Fragment Synthesis**: The circuit synthesis pass first synthesizes a circuit fragment that implements each dynamical system relation. This procedure uses a novel tableau-based search algorithm that incrementally builds the circuit fragment by successively unifying analog blocks with goals in the tableau. The unification algorithm uses a sophisticated algebraic rewrite engine capable of identifying non-trivial usages of analog blocks. This rewrite engine enables the unification algorithm to configure complex analog building blocks to implement the desired expressions.

- **Assembly**: The circuit synthesis pass then assembles all of the produced fragments to form a completed circuit. The circuit assembly procedure inserts blocks when necessary to copies and convert signals. The output of this procedure is a circuit that implements the dynamical system.

- **Place and Route**: The circuit synthesis pass then maps all blocks to the circuit to locations on the analog device and all connections in the circuit to digitally settable connections in the analog device. The placement and routing algorithm intelligently uses the device layout information to map the circuit to the analog hardware. This algorithm inserts routing blocks when necessary to make the desired connections.

1.4.8 Circuit Scaling (Chapter 9)

I provide a rigorous description of the circuit scaling pass. The circuit scaling pass computes a scaling transform for a provided unscaled ADP.

I first introduce the combinatorial geometric programming (CGP) and geometric programming problem (GP) formulations used by the circuit scaling pass. These problem formulations both encode the circuit scaling problem. The geometric programming problem (GP) is a type of optimization problem that can be efficiently

solved with a convex solver to minimize some objective function. The CGP is an extension to the convex optimization problem that supports the specification of discrete constraints over integer variables. The CGP becomes a GP once all of the integer variables are assigned to values. The circuit scaling pass uses both of these problem formulations to scale the circuit.

I then formally descibe the circuit scaling pass. I first introduce all the new notation used in this chapter. This chapter reuses the notation introduced in Chapter 8 and introduces new constructs which capture physical behaviors and limitations present in the analog device.

I then formally describe how the compiler derives a CGP from the unscaled ADP. The compiler derives a collection of linear constraints from the (nonlinear) dynamics and structure of the circuit described in the ADP. The CGP encodes the device operating range and frequency restrictions and the effects of noise and quantization error on the computation. The CGP also ensures that the scaling transform preserves the original dynamics of the unscaled ADP (and dynamical system). Specifically, the CGP ensures that the scaled dynamics at every input and output port equals the unscaled dynamics of the port times some constant coefficient. The compiler uses the delta model information from Chapter 5 to produce a scaling transform that preserves the original dynamics in the presence of manufacturing variation-induced behavioral variations. The CGP formulation enables the compiler to reconfigure blocks to better scale the circuit. The compiler encodes these reconfiguration operations as discrete variables and constraints in the CGP. The discrete constraints capture the effect of each reconfiguration operation on the scaling problem.

I then describe how the compiler uses the CGP to scale the circuit. The compiler first solves the CGP to produce a set of block reconfiguration operations which are then applied to the unscaled ADP. The compiler then concretizes the discrete CGP variables which capture the reconfiguration operations to produce a set of GP constraints. The compiler then constructs the GP by combining the derived GP constraints with a user-provided scaling objective function that captures the circuit property to optimize. The compiler solves the GP to identify the scaling transform

that best minimizes some criteria. In this thesis, I use a `balanced` scaling objective function that jointly maximizes the execution speed and the dynamic range of the scaled signals.

1.4.9 Results (Chapter 10)

I evaluate the efficacy of the compiler. I compile the twelve benchmarks presented in Chapter 3 to the HCDCv2 with the compiler presented in this thesis. I perform the following analyses to evaluate the efficacy of my compiler:

- **Power, Energy, and Quality Analysis**: I study the execution time and energy usage of each application and qualitatively and quantitatively examine the agreement between the collected analog waveforms and the ground-truth dynamical system dynamics. I find that the produced scaled ADPs execute in 0.25-1.92 milliseconds, consume 0.10-5.09 μJ of energy, and report 3.46×10^{-6}%-1.96% error. Here, the reported error is the root-mean-squared error of the measured waveform with the recovery transform applied relative to the amplitude of the reference waveform. I compute the reference waveform for each benchmark application by simulating the dynamical system with a high precision digital ordinary differential equation solver. After all compiler optimizations are applied, the collected analog waveforms with the recovery transform applied are visually indistinguishable from the reference waveforms.

- **Effect of Compiler Optimizations**: I study the effect of different compiler optimizations on the fidelity of the produced waveforms. Both the scaling transform and the partial block reprogramming feature employed by the circuit scaling pass are crucial for obtaining a good quality result for a number of the benchmarks. I also investigate the importance of incorporating delta model information into the scaling procedure and find that this improves the fidelity of the result for 10 of the 12 benchmarks. I then investigate the effect of the calibration strategy on the quality of the produced results. For 9 of the 12

benchmarks, the co-designed calibration strategy delivers comparable or lower-error results more consistently than the traditional strategy.

- **Compilation Outcomes**: I present the compilation times and study the optimality of the produced unscaled and scaled ADPs. I find that the unscaled ADPs rarely use more special-use routing and assembly blocks than necessary. The scaled ADPs frequently run at the maximum speed of the device. Generally speaking, the compiler is not able to simultaneously maximize all of the signals and values in a given ADP. However, the compiler can maximize at least one value for all benchmarks and one signal for 9 of the 12 benchmarks. For all of the benchmarks, the compiler can produce a scaled ADP in which at least half the signals occupy 50% of the port operating ranges. These results indicate that the compiler can effectively scale the ADP to attain good speeds and signal dynamic ranges while respecting operating range and frequency limitations. I then investigate how well the scaled ADPs minimize the `balanced` scale objective function introduced in Chapter 9. I find that the compiler can identify multiple scaling transforms that scale signals in different ways but attain comparable `balanced` scale objective values. These findings demonstrate that the compiler can identify multiple good scaling transforms within the space of physically viable, recoverable scaling transforms. I then investigate why the unscaled ADPs cannot be directly executed on the analog device. I find that in all cases, the unscaled ADPs violate the frequency and operating range restrictions imposed by the device.

- **Compilation Outcomes and Result Quality**: I investigate the relationship between the scaled ADP characteristics and the end-to-end result quality. The goal of this analysis is to determine if there are any ADP characteristics that are strongly predictive of the end-to-end result quality. I find that the block instance selection has a profound impact on the quality of the produced waveforms. I use this observation to inform a potential future research direction in the concluding chapter of this thesis (Chapter 11). I also find that the `balanced`

objective function value is predictive of the end-to-end result quality for 11 of the 12 benchmarks (Pearson coefficient > 0.5). For 6 of the 12 benchmarks, the `balanced` objective function value strongly correlates with the quality of the produced waveform (Pearson coefficient > 0.9). The results of the analysis can be used to inform future compilation techniques.

- **Alternative Scaling Objective Functions**: I investigate the potential of an alternative scaling objective function. This analysis aims to determine if the scaling objective function deployed by the compiler can be further improved in future work. I find that for 5 of the 12 benchmarks, the alternate scaling objective attains better or comparable quality results while consuming less energy. I use the results of this analysis to inform future research directions in the concluding chapter of this thesis.

  Realtime Case Studies: I compile and execute the real-time dynamical system applications from Section 4.13 on the HCDCv2. I find that for both dynamical systems, the HCDCv2 performs the desired signal processing operation in real-time on an externally provided signal.

1.5 Reading Strategies for this Thesis

I next present a collection of strategies for reading this thesis. These reading strategies outline which chapters to focus on and are organized by reading goal:

- *I want to understand this research area at a high level* – Read the background chapter (Chapter 2), Section 3.1 of the dynamical systems chapter (Chapter 3), the introduction and Sections 5.1-5.3 of the reconfigurable analog devices chapter (Chapter 5), and the conclusion chapter (Chapter 11). These sections provide an overview of prior work, outline the high-level challenges for the application domain and target hardware, and outline some productive future research directions for this domain.

- *I want to target a differential-equation solving analog hardware* - Read the dynamical systems chapter (Chapter 3), dynamical system applications chapter (Chapter 4), reconfigurable analog devices chapter (Chapter 5), and the analog device programs chapter (Chapter 6). These chapters rigorously describe the behavior of the target hardware platform and provide a set of twelve worked examples for the HCDCv2 analog hardware.

- *I want to compare against your results* - Read the dynamical systems chapter (Chapter 3), dynamical system applications chapter (Chapter 4), and results chapter (Chapter 10). These chapters rigorously describe the benchmark applications and present a rigorous evaluation of the compiler on the target hardware platform.

- *I want to understand how the compiler works at a high level* - Read the dynamical systems chapter (Chapter 3), the introduction and Sections 5.1- 5.6 of the reconfigurable analog device chapter, section 6.1 of the unscaled/scaled ADPs chapter (Chapter 6), and the compilation overview chapter (Chapter 7). These sections introduce the specification languages employed by the compiler and provide an overview of the operation of the compiler on an example program.

- *I want to implement or extend the compiler* - First read the dynamical system chapter (Chapter 3), the reconfigurable analog devices chapter (Chapter 5), and unscaled/scaled ADPs chapter (Chapter 6). These chapters provide a comprehensive overview of the compiler inputs and outputs. Then read the compilation overview (Chapter 7), circuit synthesis (Chapter 8), and circuit scaling (Chapter 9) chapters. These chapters rigorously describe the operation of the compiler.

1.6 Summary

Specialized computing platforms, implemented on a range of devices including analog, photonic, and digital devices, are becoming pervasive and crucial for satisfying the

computational needs of different domains. We are already seeing a proliferation of specialized devices that efficiently solve problems in machine learning, signal processing, and biology. This thesis focuses on an emergent class of reconfigurable ultra-low power analog devices which solve dynamical systems.

A dynamical system is a system whose state evolves over time. Dynamical systems appear in a wide variety of fields including mathematics, physics, chemistry, biology, economics, engineering, and medicine. Typically, practitioners are interested in simulating a dynamical system. Digital dynamical solvers have difficulty efficiently simulating non-linear differential equations or differential equations with dynamics which operate on different time scales. These efficiency issues are exacerbated for dynamical system applications which process realtime signals or execute on resource-constrained embedded systems.

In recent years, there has been a proliferation of ulta-low power reconfigurable analog devices which solve dynamical systems. These analog devices use standard CMOS processes and leverage transistor physics to perform computation. These devices are attractive computational targets because they consume very little power and deliver predictable performance regardless of the size and complexity of the dynamical system. In these systems, the time required to run a dynamical system can be computed statically at compile-time. These modern analog computing platforms are inherently approximate and offer medium-precision blocks that introduce error into the computation. Furthermore, these blocks cannot be manufactured at low tolerances and therefore experience a high degree of variation post-fabrication.

Research Problem: Presently, these devices lack software tooling and must be manually configured by the programmer to implement the desired dynamical system computation. These devices are programmed by routing together configurable analog blocks using programmable interconnects. To faithfully implement a computation, the device physics must preserve the original dynamical system dynamics such that the practitioner can recover the original dynamical system variable trajectories from the measured signals at runtime by applying a recovery transform.

This class of analog devices present fundamentally different programming chal-

lenges than digital computing platforms. First, low-level physics of the device has a fundamental effect on the computation and must therefore be taken into account when programming the device. Analog blocks impose frequency, current, and voltage range limitations and have unique noise and error characteristics. All of these low-level physical behaviors must be taken into consideration when programming the device.

Analog blocks are also subject to process variation-induced behavioral deviations. While researchers have developed hardware-based mitigation mechanisms for dealing with process variation (such as device calibration), these mitigation techniques do not adequately eliminate all unwanted behaviors across all blocks. Any behavioral deviations that cannot be eliminated with hardware mitigation techniques must be compensated for in software or contribute to the computation error.

These analog devices offer highly specialized programmable analog blocks that implement anything from simple functions to sets of differential equations. The set of available analog blocks may vary wildly from device to device. Furthermore, these analog blocks cannot be arbitrarily routed together since the routing environment in such devices is highly restrictive.

This Thesis: The goal of this research is to automate the programming process so that these devices are more accessible to programmers. In this thesis, I present a compiler that takes as input a dynamical system and produces as output a configuration for the analog device which implements the dynamical system on the analog hardware. The compiler first automatically derives an analog circuits comprised of configured analog blocks which implement the target dynamical system. The compiler then scales all the values in the provided unscaled ADP and produces a scaled circuit respects all of the physical constraints and behaviors present in the device. The transformed circuit preserves the original dynamics of the dynamical system – that is, the original dynamics of any signal can be recovered at runtime by multiplying it by a statically derived constant factor. This transform compensates for the process variation-induced behavioral deviations that could not be corrected for in calibration and rescales the signals and values in the circuit to reduce the effect of noise and

quantization error.

In this thesis, I develop a compilation toolchain which targets the ultra-low-power HCDCv2 analog device and use the compiler to map twelve dynamical system benchmark applications to a physical HCDCv2 device. To the best of my knowledge, this compiler is the first to successfully target a physical (as opposed to simulated) programmable analog device for dynamical systems and this thesis is the first to present experimental results for any compiled computation executing on any physical programmable analog device of this class.

Chapter 2

Related Work

This chapter presents an overview of the relevant related work for the work in this thesis. In this chapter, I cover the following topics:

- **Dynamical Systems (Section 2.1)**: I introduce application domains for dynamical systems and provide an overview of the types of dynamical systems commonly seen in practice. I then discuss the numerical methods used for simulating dynamical systems and their drawbacks.
- **History of Analog Computing (Section 2.2)**: I provide a brief overview of how practitioners historically used analog computers to simulate dynamical systems. I describe the computational model and programming model used for these historical analog computing platforms. I then provide an overview of the automated programming techniques used for these hardware platforms.
- **Dynamical System-Solving Reconfigurable Analog Devices (Section 2.3)**: Recently, researchers have proposed modern reconfigurable analog devices which leverage the analog behavior of transistors to execute dynamical system computations. These devices can be configured post-fabrication to simulate a variety of different dynamical systems. I provide an overview of how these devices simulate dynamical systems, why they are attractive computational targets, and how they are programmed today. I describe how the compiler presented in this thesis lowers the barrier of entry for programming this class of devices.

- **Other Reconfigurable Analog Computing Platforms (Section 2.4)**: Researchers have proposed a multitude of reconfigurable mixed-signal and analog computing platforms that target a variety of other application domains. I provide an overview of these computing platforms, broken up by application domain. I contrast the programming techniques used to target each class of computing platforms with the compilation techniques proposed in this thesis.

- **Software Techniques (Section 2.5)**: I provide an overview of related synthesis, compilation, hardware configuration, and numerical computing techniques. I describe how each software technique relates to the compilation techniques employed by my compiler.

2.1 Dynamical Systems

A dynamical system is a system whose state evolves over time. Dynamical systems appear in a wide variety of fields and are typically implemented as ordinary differential equations or partial differential equations. Typically, the goal is to simulate or solve a dynamical system. Today, digital differential equation solvers are used to simulate dynamical systems. Many of these solvers have difficulty efficiently solving non-linear systems or systems with dynamics that operate on different time scales. To ameliorate these issues, practitioners often introduce approximations into the dynamical system. Common approximations include linearization and replacing parts of the dynamical system with closed-form approximations. This section provides an overview of dynamical system applications and types of dynamical systems and discusses the numerical simulation approaches used to simulate dynamical systems today.

2.1.1 Applications

Dynamical systems appear in a wide variety of fields including mathematics, physics, chemistry, biology, economics, engineering, and medicine. In the biological sciences, practitioners use dynamical systems for medical dosage optimization and disease pre-

diction applications and to better understand biological phenomena. [74, 117]. In the physical sciences, practitioners use dynamical systems to model physical phenomena such as earthquakes, glacial movement, and planetary motion [47, 112]. In economics, practitioners use dynamical systems to predict changes in the market [118]. These dynamical system computations are typically run on compute clusters and workstations where power is readily available.

Dynamical systems are also deployed on embedded systems which interact with the environment. These dynamical systems process signals, from sensors, for example, to reconstruct higher-order information from the environment in real-time [78, 14]. Embedded dynamical systems can also produce driving signals to control actuators such as motors to meet some sort of objective [33, 9]. This sort of use case is common in fields such as robotics. These sensor-actuator applications have real-time performance requirements and are often implemented on resource-constrained embedded systems such as microcontrollers.

Dynamical systems can also be used to perform both unconstrained and constrained optimization. Researchers have proposed procedures for transforming optimization problems into time-varying dynamical systems [20, 143]. These dynamical system encodings of optimization problems converge to a local optimum over time, given an initial guess. It is also possible to encode various computer algorithms with dynamical systems. Researchers have previously presented dynamical systems which sort lists, diagonalize matrices, and solve SAT problems [19].

Dynamical systems also appear in spiking neural networks. A spiking neural network is an artificial neural network inspired by biological neuronal networks. A spiking neural network contains a collection of interconnected neurons which communicate with one another. Each neuron produces a spike as output when its membrane electrical charge surpasses a certain threshold. The behavior of the membrane electrical charge over time is modeled with a time-varying dynamical system.

Dynamical systems also appear in some classes of machine learning models. Continuous-time recurrent neural networks and neural ODEs are both implemented with dynamical systems [45, 12, 25].

Relationship to this work: In this thesis, I present a compiler that maps dynamical systems to dynamical system-solving reconfigurable analog devices. These devices are attractive computational targets for real-time embedded dynamical system computations since they operate at ultra-low power and have predictable performance characteristics. Some of these devices are also able to interface with sensors and actuators directly. Refer to Section 2.3 for an overview of dynamical system-solving reconfigurable analog devices.

2.1.2 Types of Dynamical Systems

A dynamical system typically consists of one or more interdependent variables which change over time. Dynamical systems are usually implemented as a system of ordinary differential equations (ODEs) or partial differential equations (PDEs). Systems of ordinary differential equations contain derivatives taken with respect to an independent variable (usually time). Partial differential equations contain derivatives taken with respect to other variables.

There are various subclasses of ODEs that support non-trivial operators. Stochastic differential equations (SDEs) are systems of ODEs with stochastic behavior [8]. These stochastic systems typically have ODEs with a deterministic component and a state-dependent stochastic component. Delay differential equations (DDEs) are systems of differential equations that contain time delayed variables [73]. A time-delayed variable is a variable which references a value in the past. Because both of these subclasses of ODEs have difficult-to-simulate dynamics, they typically require specialized solvers.

Relationship to this work: In this thesis, I focus on time-varying dynamical systems made up of ordinary differential equations. The dynamical systems I target in this work do not contain stochastic behavior or time-delayed variables.

2.1.3 Differential Equation Solvers

Practitioners simulate dynamical systems comprised of ordinary differential equations on digital hardware with ordinary differential equation (ODE) solvers. Classic ODE solver methods operate by breaking up simulation time into multiple time steps and then sequentially computing the state of the dynamical system at each time step [22, 10]. The exact time step segmentation strategy depends on the solver. Some solvers accept a fixed time step size from the user, while other solvers adaptively tune the time step size depending on the system dynamics. One drawback to this simulation technique is that it cannot capture any dynamics which occur between time steps [113, 114]. As a result, the simulator may produce inaccurate results depending on the characteristics of the dynamical system and how the solver is parametrized.

Finite difference methods approximately solve ordinary differential equations by approximating derivatives with value differences [77]. These methods require the dynamical system to be linearized before simulation. This linearization step eliminates non-linear dynamics from the system and reduces the fidelity of the end-to-end result. Finite difference methods are popular because they are easy to implement and can be solved efficiently in more resource-constrained systems.

Some kinds of ordinary differential equations are difficult to simulate accurately and performantly with digital ODE solvers. Dynamical systems with both fast-evolving and slow-evolving dynamics can introduce numerical instabilities into the digital simulation [94, 36]. This can cause the ODE solver to produce an inaccurate result. Practitioners will often substitute fast-evolving dynamics with closed-form solutions when possible to resolve instability issues in the simulation[107, 41, 133]. These closed-form solutions often involve computationally expensive, highly non-linear operators [30, 27].

Non-linear dynamical systems are often difficult to simulate as they often use expensive operators and are difficult to analyze automatically. Non-linear dynamical systems also introduce non-linearities into the dynamical system dynamics. These non-linearities may introduce numerical instabilities into the digital simulation. Prac-

titioners often linearize dynamical system non-linearities to make the computations tractable and amenable to efficient digital simulation [58, 98, 99, 17]. For dynamical systems which model physical phenomena, this reduces the accuracy of the model in relation to the corresponding physical system [37, 138, 21].

Practitioners use a variety of specialized solvers to simulate dynamical systems implemented with PDEs, SDEs, and DDEs. Dynamical systems comprised of partial differential equations are typically solved with finite difference or finite element methods [7]. Variants of systems of ODEs, such as SDEs and DDEs, are simulated with specialized solvers [96, 13]. These specialized solvers efficiently simulate the difficult-to-simulate behaviors in these systems.

Relationship to this work: In this thesis, I focus on simulating dynamical systems with analog hardware. Under this paradigm, the evolution of the currents and voltages within the analog device capture the dynamical system's dynamics over time. This computational model does not discretize simulation time and can operate on non-linear dynamical systems. Therefore, this analog approach to solving dynamical systems does not suffer from the same set of issues as digital ODE solvers.

2.2 History of Analog Computing

Historically, researchers used electrical analog computers to perform dynamical system simulation and study control systems [130, 34, 104, 91]. Practitioners used these analog computers to perform flight simulations, design autopilot systems, implement radar systems, and teach control theory [130, 34, 104, 91]. Digital computers eventually replaced these analog computing platforms as transistor sizing improved.

Early electrical analog computers primarily performed computation with analog voltages and provided components that implemented basic mathematical operators [134, 63, 64, 127]. Common operators included simple summation, summation with integration, scaling by a constant value, and signal multiplication. These components implemented high-precision analog primitives that could be manufactured at low tolerances. As a result, these components performed computation at high ac-

curacy – usually within 0.01% error relative to the full-scale range of the signal for linear components [65].

Programming Techniques: Practitioners routed these components together to form circuits through the use of a patchbay. These early analog computers contained programmable switches that changed the functions implemented by the hardware blocks and potentiometers that scaled signals by constant coefficients[49, 63]. Practitioners would set the potentiometer values and manipulate the switches on the analog computer to program the components to deliver the desired behavior. These computers also supported fine-grain monitoring of each computational unit's input and output signals. Practitioners would leverage this capability to identify instances where signals are saturated and to more easily debug the programmed circuit [53].

To program a computation to the analog computer, the practitioner would first manually derive a circuit comprised of configured components that implements the desired dynamical system computation with pen and paper. After identifying a circuit, the researcher would then scale the potentiometer values in the circuit so that the computation executes accurately without saturating any of the circuit components [6]. The practitioner would then translate the derived circuit to a set of potentiometer values, block switch settings, and a patchbay wiring scheme [49].

Relationship to this work: This thesis research focuses on modern incarnations of electrical analog computing platforms which solve dynamical systems. Unlike historical analog computers, modern analog computers are silicon chips with digitally programmable interconnects and values. These modern analog computers deliver significant energy savings but offer lower precision approximate computational blocks (1%-5% error) [132, 51, 61]. These blocks are subject to the effects of process variation and noise. The programming techniques used to configure high precision historical analog hardware would not necessarily deliver good results on modern analog hardware designed to execute approximate computations.

Modern reconfigurable analog devices perform computation primarily with analog currents and offer different programming and debugging interfaces than historical analog devices. The circuit topology is set with programmable interconnects instead

of a patchbay, and the individual components are configured by setting bits instead of configuring switches and potentiometer values. This programming interface is completely digital and more amenable to automatic configuration than historical analog devices. Historical analog devices supported monitoring all voltages which are accessible through the patchbay. In contrast, modern reconfigurable analog devices offer a limited debugging interface where only a select subset of components and signals can be monitored at runtime. Modern analog devices therefore cannot support programming techniques that monitor all signals at runtime.

2.2.1 Compilers for Historical Analog Computers

In the past, researchers developed compilers which targeted early analog computers[49, 90, 75, 43, 40, 90, 75, 55, 126, 123, 122, 90, 75, 100]. The three most complete efforts to build a compiler for an analog computer were the APACHE system [49] which targeted the PACE 231-R analog computer[63], the HAL system [43] which targeted the EAI 680 analog/logic computer [64], and the Hytran and HOI systems which targeted the Hydac 2400 and EAI 8900 analog computers. Several other early compilation approaches were aspirational and never fully implemented [55, 126], failed to fully automate the compilation process [123, 122, 40, 90, 75], or targeted idealized hardware platforms which did not exist in the real world [100].

The APACHE compiler automatically derived patchbay wiring instructions, potentiometer settings, and switch configurations from the high-level dynamical system specification [49]. The mapping technique employed by the APACHE system was highly specialized to the PACE-231-R and could not be generalized to other analog computing platforms or extended to work with mixed-signal or logical blocks common in hybrid computation [42]. The HAL compiler automatically derived digital programs, patchbay wiring instructions, potentiometer settings, and switch configurations from a low-level assembly program. The hybrid assembly programming language offered analog component-based primitives that provided programmers with fine-grain control of the produced circuits [43]. The HAL compiler worked with programs provided at a much lower level of abstraction than the APACHE compiler

but supported the digital and mixed-signal blocks often seen in hybrid computing platforms. The Hytran and HOI systems [40, 90] were extensively used to check programs statically but did not automate the circuit generation and circuit scaling process. These systems required the end user to manually provide the circuit topology and scale factors to the software system. In summary, of the fully realized compilers, only the APACHE compiler presented an approach for automatically deriving a circuit from a dynamical system.

The APACHE and HAL compilers supported automatically scaling the target computation to respect the operating ranges present in the hardware [49]. These approaches derived constant scaling factors that were then multiplied with the potentiometer values to scale the target computation to respect the hardware operating ranges [52, 53, 93]. These scaling techniques leveraged a hardware-in-the-loop scale factor refinement algorithm that dynamically adjusted the scale factors when signals are saturated. These scaling techniques monitored the output voltage of each component to refine the scaling transform. In some cases, these automated scaling procedures produced scaling transforms with large scaling factors – in these cases, the practitioner had to adjust the circuit topology manually. These early automated scaling techniques only worked with circuits comprised of linear operators and multiplication operations, only considered the operating range restrictions present in the hardware, and focused on producing valid transforms rather than optimal transforms.

Relationship to this work: The automated circuit generation procedure employed for the APACHE compiler cannot be directly applied to this modern analog hardware presented in this work. The APACHE compiler was highly specialized to target the PACE 231-R analog computer lacked support for mixed-signal and digital components and only supported voltage-mode computation. The modern analog hardware targeted in this work computes with analog currents and contains digital blocks such as LUTs and analog-digital interfaces such as DACs and ADCs – the APACHE compiler does not support this computational model or these blocks. Moreover, the APACHE compiler deployed a mapping algorithm that was not readily generalizable or extensible and could not identify creative usages of components. In contrast, the compiler

presented in this work can leverage algebraic rewrite systems and synthesis techniques to identify non-trivial compositions of blocks. Other early compilers could not automatically generate circuits and required the end user provide the circuit topology as an input.

The automated scaling procedures used for early analog computers cannot be applied to the modern dynamical system-solving analog computers targeted in this work. Early scaling methods only reasoned about operating range limitations and focus on producing a valid (often suboptimal) scaling transform – this is insufficient for targeting modern analog hardware. Unlike historical analog devices, which offer high-precision low-tolerance computational blocks, modern analog devices offer approximate computational blocks that introduce more error into the computation and are more sensitive to process variation and noise. The scaling approach employed in this thesis considers a wide range of behaviors, including process variation, noise, quantization error (from digital logic), and frequency limitations. This scaling approach also scales the circuit to optimize a circuit characteristic, such as dynamic range or execution speed. These capabilities enable the compiler presented in this thesis to map the high-level computation to approximate computational blocks more effectively and more productively attenuate away unwanted analog behaviors.

The scaling algorithm presented in this thesis is also capable of automatically reasoning about a wide range of non-linear operators, including exponentiation, transcendental functions, and discontinuous functions. This scaling approach can also reprogram the analog blocks to better scale the circuit. In contrast, early automated scaling approaches require user intervention to reprogram the hardware and only scaled circuits made up of linear operators and signal multipliers.

Another shortcoming of early automated scaling approaches is that they often were hardware-in-the-loop techniques that monitored all component output voltages. It is not feasible to monitor all intermediate signals on a modern analog computing platform implemented in silicon. Typically, these devices offer only a few externally accessible input and output interfaces. The automated scaling procedure presented in this thesis does not require any runtime monitoring of signals to scale the circuit.

The scaling transform is computed entirely at compile-time from specifications of the target hardware and dynamical system.

2.3 Dynamical System-Solving Reconfigurable Analog Device

One prominent line of work focuses on reconfigurable analog devices that solve dynamical systems [105, 29, 128, 31, 140, 132, 51, 61, 141]. A dynamical system-solving analog device is a type of reconfigurable analog device that uses the analog behavior of transistors to perform dynamical system simulation. These analog devices leverage the advanced metal–oxide–semiconductor fabrication technologies traditionally found in consumer electronics such as cell phones, biomedical devices, and edge devices [132].

These reconfigurable analog devices operate in an ultra-low power regime and can efficiently execute potentially complex non-linear dynamical system computations with microwatts to milliwatts of power. Because these reconfigurable analog devices represent dynamical system quantities with continuously evolving currents and voltages, they are not subject to the time discretization errors discussed in Section 2.1.3. This class of devices also offers predictable performance characteristics. The compiler can compute the correspondence between wall-clock (execution) time and dynamical system time statically at compile-time for such platforms. Typically, these devices can simulate a one-time unit of dynamical system time in 7.93 microseconds to 0.30 milliseconds of wall-clock time, depending on the hardware platform. Because these devices execute perform computation in parallel, the performance of the hardware does not necessarily degrade with problem size.

Some of these analog computing platforms have special capabilities. For example, the devices may be capable of directly processing analog signals in real-time [132, 51, 61] or directly implementing stochastic computation with analog noise [140]. These performance characteristics and hardware capabilities together make these devices especially appealing for real-time embedded dynamical system computations.

Internally, these devices contain collections of digitally configurable analog blocks that may be routed together with digitally programmable interconnects to form various analog circuits. The goal of compilation is to identify a circuit that preserves the original dynamical system dynamics — that is, the original dynamical system dynamics can be recovered at runtime from the signal trajectories with a recovery transform. The computation is then run by powering on the device and observing the evolution of the currents and voltages of interest over time. The original dynamical system variable trajectories are then recovered from the voltage and current trajectories at runtime by applying a recovery transformation. Because these devices directly exploit the physics of the underlying hardware substrate, these computations are susceptible to the effects of analog noise and process variation. The computation must also respect the analog device's operating range and frequency restrictions to execute as expected on the analog device.

The computational model described above is a continuous-time computational model which directly maps dynamical system simulation time to wall-clock time. Researchers have also proposed hybrid computing platforms which solve dynamical systems in discrete-time using finite-difference methods [79]. These hardware platforms use analog circuitry to compute the derivatives of the dynamical system variables with respect to time. In this thesis, I focus on continuous-time dynamical system-solving analog devices.

Dynamical system-solving analog devices can be used to execute a variety of different computations. Researchers have developed reconfigurable analog devices which solve SAT problems and perform constrained optimization [142, 137]. This research formulates the target SAT problems and quadratic programming problems as time-varying dynamical systems.

Programming Techniques: Modern dynamical system-solving analog devices are typically programmed directly with little or no automation [50]. To program these devices, practitioners configure individual blocks on the device and enable the necessary programmable interconnects to form the desired circuit. Before writing the circuit to the device, the programmer may have to manually transform the circuit parameters

to ensure the circuit respects the device's operating range and frequency constraints. The designer may also transform the circuit to reduce the effects of noise and process variation. After transforming the circuit, the programmer manually derives the recovery transformation. This recovery transform recovers the original dynamical system dynamics from the signal trajectories. The programmer must carefully choose a circuit transformation that ensures the original dynamical system is recoverable at runtime.

Relationship to this work: This thesis research presents a compiler that targets programmable analog devices that solve differential equations in continuous time. I designed this compiler to be broadly applicable across multiple different hardware platforms and dynamical system use cases. The compiler works with a dynamical system specification and a hardware specification that describes the blocks and connections available on the analog device. The compiler works with these specifications to produce an analog circuit – this enables the compiler to target a variety of reconfigurable dynamical system-solving analog devices and compile dynamical systems from a variety of different application domains. I demonstrate that this compiler can effectively target a fabricated reconfigurable dynamical system-solving analog device [132, 51, 61] in practice in this thesis.

2.3.1 Compilers for Dynamical System-Solving Analog Devices

There has been some recent work on mapping biological networks to programmable analog devices specifically designed to run biological systems. The proposed compiler maps biological models implemented in a systems biology markup language (SBML) to a reconfigurable analog device that simulates reaction networks [140]. The compiler produces, as output, a configuration for the target hardware and a system of differential equations that implements the biological model. The target device contains twenty block instances of a chemical reaction block that can be configured to implement various chemical reactions.

The key technical contribution in this work is a program transformation that expands lumped kinetic models into systems of chemical reactions. This program transformation operates directly on the biological model. A lumped kinetic model condenses systems of chemical reactions into a set of closed-form algebraic functions which capture the steady-state of the relevant compounds in the system. The authors present a pattern matching-based method for expanding the lumped kinetic models. The expanded reactions are then directly mapped onto the target analog hardware.

Relationship to this work: The work presented in [86] was published after our work on analog compilation [4, 2, 3]. The lumped kinetic expansion optimization and SBML parser presented in [86] is complementary to my own work and can be incorporated into my compiler as a simple, domain-specific front-end. In this scenario, the differential equations emitted by the lumped kinetics expansion optimization pass would become an input to my compiler.

The compiler presented in [86] does not systematically reason about the low-level physical behaviors in the hardware (such as process-variation induced behavioral variations) or transform the circuit to improve the dynamic range of the signals. While the authors perform some corrections to handle unexpected gains in analog currents produced from digitally settable data fields, these corrections are localized to the specific data fields. These corrections therefore do not compensate for all unexpected gains in the device. As a result, the compiled computation produces results that do not agree with the reference implementation of the program. The authors list signals with small dynamic ranges and unexpected gains as possible sources of imprecision in the compiled computation.

In constrast, the compiler presented in this thesis automatically transforms the produced circuits to respect all of the operating range and frequency limitations present within the hardware and considers the effects of process variation, quantization error, and noise on the computation. The compiler also increases the dynamic ranges of the signals to reduce the effect of error on the computation. As a result, the compiled computations produce results that strongly agree with the reference implementations of the benchmark applications.

The circuit generation algorithm presented in [86] uses a hand-implemented mapping procedure to map expanded reactions to analog circuits. This mapping procedure is designed to target the cytomorphic chip [140] specifically and cannot be generalized to other kinds of reconfigurable dynamical system-solving hardware platforms. In contrast, the compiler presented in this thesis targets a hardware specification and offers a general compiler architecture for targeting a variety of different reconfigurable differential equation-solving analog devices.

In summary, the compiler presented in [86] can only be compared to a subset of the work presented in this thesis (circuit synthesis) and is unlikely to be nearly as broadly applicable as the techniques presented in this work. Moreover, parts of the compiler presented in [86] which deal with the analog behavior are still relatively immature when compared to the compiler presented in this thesis.

2.4 Other Kinds of Reconfigurable Analog Devices

Pure and mixed-mode analog accelerators have been developed for accelerating a broad range of applications, including neural networks, SAT solvers, and neuromorphic computations [15, 109, 102, 62, 87, 89, 38]. Modern reconfigurable analog devices may leverage a variety of different physical phenomena to implement computation. Researchers have proposed photonic analog computing platforms which leverage the physical properties of light to perform computation, for example [85, 116, 81]. In this section, I focus on electrical analog devices implemented with mixed-signal and analog ICs. This class of reconfigurable analog devices targets a wide variety of application domains and computational models. In this section, I introduce reconfigurable analog devices by application domain.

2.4.1 Spiking Neural Networks

Researchers have designed analog computing platforms that implement spiking neural networks [89, 69, 15]. These devices are attractive computational targets because they can simulate large spiking neural networks (SNNs) at low power. Typically, these

platforms implement the neuronal models with analog circuitry and route/process spikes to neurons with digital circuitry. Researchers have also designed mixed-signal SNN accelerators which leverage device mismatch to implement device-specific, non-uniform neuron models [89].

Programming Techniques: Compilers that target mixed-signal spiking neural networks accelerators typically directly map the target spiking neural network to hardware elements. This mapping procedure directly maps neuron parameters to circuit parameters and neurons to analog functional units within the hardware. Because mixed-signal SNN accelerators offer analog functional units which directly correspond to SNN elements, the mapping process is relatively straightforward. The compiler employed in [89] maps a non-linear dynamical system to a network of analog neuron elements. This mapping identifies a neuron topology that approximately implements each differential equation in the target dynamical system. With this computational model, integration is performed in discrete time.

Relationship to this work: This thesis targets dynamical-system solving analog devices that solve differential equations in continuous time. This compilation for this class of hardware is fundamentally different from the compilation problem for spiking neural networks. For our target class of hardware, there is no straightforward mapping between the target dynamical system and the analog computational units provided by the hardware. Furthermore, the devices targeted in this thesis perform dynamical system simulation purely in analog. The compiler must explicitly introduce logic to re-use and route signals. In contrast, mixed-signal SNN accelerators represent neuron inputs and outputs as digital signals. The SNN compiler is free to route and buffer these digital signals through the chip without reasoning about any analog behavior.

2.4.2 Neural Networks and Machine Learning

Researchers have also developed mixed-signal programming-in-memory systems which implement multiply-accumulate operations directly in memory with analog ICs [48, 23, 24, 88]. These systems are attractive computational targets for big-data applications because they perform computation directly on data in memory and therefore

require less data movement. The stored data is typically binary data in these systems, and the applied weights are typically decimal values between -1 and 1.

Researchers have also developed mixed-signal and analog accelerators which fully implement neural networks [111, 39, 121]. Some of these mixed-signal devices implement neural networks which approximate digital sub-computations [39, 121].

Programming Techniques: The analog logic in these accelerators typically implements straightforward mathematical sub-computations such as weighted summation. These analog logic units are typically embedded in a larger digital system that implements the rest of the neural network operators. Depending on the hardware platform, the analog units are programmed through specialized hardware instructions or automatically programmed by the compiler during the mapping process. Because there is a direct correspondence between the neural network topology and the functional units on the device, the mapping process is relatively straightforward.

One key challenge with programming analog MAC computational units is accurately setting the weights. Because analog ICs are sensitive to low-level physical behaviors present in the hardware, the programmed weights do not accurately reflect the actual weights applied in the hardware. Researchers have proposed alternate neural network training algorithms that consider hardware imperfections when training the models [24].

Relationship to this work: This thesis targets dynamical-system solving analog devices. These devices fully implement the entire dynamical system computation in analog hardware. For this hardware, there is no straightforward mapping between the target dynamical system and the provided functional units. The compiler presented in this thesis identifies non-trivial mappings between the hardware and the dynamical system.

The dynamical systems targeted in this thesis often contain values that lie outside of the range of values supported by the analog hardware. The compiler presented in this thesis transforms the computation so that it can be accurately implemented in the analog hardware.

2.4.3 Field-Programmable Analog Arrays and Analog Fabrics

Researchers have developed reconfigurable analog fabrics which enable individual transistors to be routed together [120, 56, 54, 11, 108]. These versatile devices are capable of implementing a variety of different analog and mixed-signal circuits.

Programming Techniques: These devices are programmed with a Simulink-style block diagram language. The configurable blocks in this language are parametrized analog circuits which route together transistors within the fabric. The associated software toolchain composes together the configured circuits in the block diagram to form a low-level circuit.

Relationship to this work: The compilation techniques presented in this thesis are complementary to the software techniques used to target field-programmable analog arrays. The compiler presented in this thesis targets analog devices with programmable analog functional units that implement high-level mathematical functions. The compilation techniques presented in this thesis can automatically derive a block diagram that implements a dynamical system, given a library of FPAA mathematical blocks. The FPAA software toolchain can then be used to map the derived block diagram to a low-level circuit.

2.5 Compilation and Synthesis Techniques

The compilation techniques presented in this thesis draw inspiration from compilation and synthesis techniques presented in academic literature. In this section, I provide an overview of related techniques. For each software technique, I describe the relationship between the proposed technique and the technique employed by our compiler.

2.5.1 Deductive Synthesis

The broad concept of a tableau has been widely used in theorem proving [1] and for the synthesis of functional programs [82, 84, 83]. In this context, logical deduction rules transform a set of assertions and goals into a proof, potentially with output

entries that make it possible to extract a program from the proof.

Relationship to this work: The compiler uses a tableau to organize a search for a configuration of the target analog hardware platform that is algebraically equivalent to the specified dynamical system. Unlike these previous approaches, the compiler works with complex multifunctional analog components, not standard programming language primitives. To correctly utilize these components, the compiler uses algebraic unification and must deal successfully with the cascading relation entanglement inherent in the use of such powerful but complex analog components.

Also, unlike these previous approaches, the compiler synthesis algorithms operate in the presence of resource constraints — they synthesize the dynamical system onto a hardware platform with finite resources. The compiler synthesis algorithms therefore must track the resources that have been consumed in the synthesis (including complex partially consumed components), with the synthesis failing if it consumes too many resources.

2.5.2 Code Generation

Code generators extract machine code from intermediate representations of the programs [35, 5, 44]. Code generators use tree pattern matching to translate code trees into sequences of machine instructions. Typically, code generators work with concise machine specifications which map IR patterns to instructions.

Relationship to this work: The compiler presented in this thesis works with dynamical systems that may contain feedback loops and circular dependencies. These dynamical systems cannot be represented as trees and cannot be mapped to analog hardware with code generation techniques.

Individual differential equation relations can be represented as trees with variables and constant values as terminal nodes. However, for these relations, the mapping between hardware blocks and differential equation terms and expressions is often highly non-trivial. The compiler may need to select and apply non-trivial transformations that alter (and sometimes complicate) the expression tree structure while mapping the differential equation to the hardware. Code generation algorithms do not support

these sorts of transformations.

2.5.3 Superoptimization and Rewrite Systems

Researchers have proposed domains-specific superoptimizers which combine mathematical and machine rewrite rules to generate search spaces of instruction sequences that implement a given computation [68, 136]. Typically the goal of such optimizers is to implement a computation with a specific instruction set. Superoptimizers typically employ search-based algorithms to find efficient instruction sequences.

Recently, researchers have developed tools for developing solver-aided languages [131]. These general-purpose tools enable programmers to define rewrite systems that transform programs.

Relationship to this work: The compiler proposed in this work differs from superoptimization research in that it produces inherently parallel analog hardware configurations with no concept of sequencing. The target blocks include a finite set of complex, potentially partially utilized analog building blocks optimized for analog efficiency, not digital machine instructions. And the relevant reasoning involves continuous, non-linear functions, not digital logic.

The compiler uses a custom, multi-stage circuit synthesis procedure to construct a circuit that implements a given dynamical system. This circuit synthesis procedure uses an off-the-shelf computer algebra system that efficiently searches over algebraic rewrite rules. The multi-stage circuit synthesis algorithm uses domain information to decompose the problem and efficiently synthesize circuits. To implement the circuit synthesis procedure with a general rewrite system, the hardware domain information, algebraic rewrite rules, and usages of the analog blocks would need to be encoded as rewrite rules. This embedding of the synthesis problem is likely to produce a prohibitively large search space which the generalized solver would have to navigate efficiently.

2.5.4 Compilers for CGRAs

Researchers have developed compilation techniques for coarse-grained reconfigurable architectures (CGRAs) [125, 97]. CGRAs are digital spatial computing platforms that typically offer processing elements and memory blocks linked together with a programmable mesh. Depending on the architecture, the individual processing elements may themselves be configurable and implement multiple operators. CGRAs are digital computational substrates and therefore support storing and loading digital values from memory.

Researchers have developed compilation techniques and hardware specification languages for this class of accelerators [70]. These contributions focus on accelerators that offer digital computing elements and offer language constructs for memories, registers, and communication interfaces. The associated compilation techniques reason about data movement and exploit parallelization opportunities. Researchers have also proposed techniques that automate the process of mapping a data-flow graph to a CGRA [26]. These techniques identify embeddings of the data-flow graph in the CGRA computational substrate.

Relationship to this work: The compilation techniques and language constructs used for CGRAs cannot be effectively applied to differential-equation solving analog computing platforms. Reconfigurable analog devices primarily work with analog signals which cannot readily be re-routed, buffered, loaded, or stored during execution. Furthermore, a key focus of CGRA compilers is in identifying parallelism. Because reconfigurable analog devices execute all computation in parallel by design, the compiler does not need to identify parallelization opportunities when mapping the computation to the hardware. The core circuit synthesis algorithm presented in this thesis solves a fundamentally different set of problems than the CGRA compilers.

The CGRA mapping technique proposed in [26] is similar to the place+route algorithm proposed in this thesis. One key difference is that this mapping technique does not consider the spatial orientation of the functional units when mapping the DFG to the hardware. In reconfigurable analog devices, spatially co-located blocks are

easier to connect together than spatially distant blocks. The place+route algorithm presented in this thesis exploits the blocks' spatial orientation to more performantly map the analog circuit to the hardware.

2.5.5 FPGA Place+Route Algorithms

Over the years, researchers have developed a multitude of place-and-route procedures for mapping digital designs to FPGAs [124, 115, 16]. FPGA place-and-route procedures map digital logic elements to physical resources on the FPGA and maps connections between elements to digitally settable paths in the FPGA. Typically, these place-and-route algorithms identify placements that minimize the length of the wires or place digital elements relatively uniformly within the computational substrate. These place-and-route procedures take advantage of the rich routing environment offered by FPGAs. For example, some place-and-route algorithms will randomly place logic elements within the FPGAs while exploring candidate digital design layouts – because FPGAs offer a large number of interconnects, these random placements likely translate to valid layouts of the design.

Relationship to this work: In this thesis, I present a place+route algorithm that maps an analog circuit to the programmable analog hardware. This place+route algorithm maps analog blocks to physical blocks on the analog device and maps connections between elements to digitally settable paths within the analog device. This place+route algorithm presented in this work differs from FPGA place-and-route algorithms because it targets a device that offers a highly restrictive routing environment which requires the allocation of specialized functional units to form certain connections. The place+route algorithm presented in this thesis generates valid block placements in the presence of restrictive routing conditions and introduces functional units when necessary to form connections. In contrast, FPGA place+route algorithms operate under the assumption that a large number of placements are valid and may explore a large number of invalid placements when applied to this class of hardware. FPGA place+route algorithms also do not typically support introducing specialized functional units to form connections.

2.5.6 Interval Analysis

Interval analysis has a long history in fields such as electrical engineering, control theory, and robotics [71, 66]. Researchers have proposed algebras for propagating intervals through mathematical functions [57, 67]. These interval analysis techniques are often used in numerical computation to statically analyze user-defined problems before solving them. Some interval analysis techniques can contract intervals to attain tighter bounds on variables and expressions. There also exist interval analysis techniques that automatically derive interval bounds for ordinary differential equations [28, 101]. For some biological and physical systems, it is possible to derive tight interval bounds analytically by leveraging conservation laws [72, 60].

Relationship to this work: The compiler presented in this thesis uses interval analysis to bound the analog signals for the parameter scaling process. The compiler uses basic interval arithmetic (without contraction) to propagate intervals through mathematical functions [57].

The compiler presented in this thesis requires the programmer to provide interval annotations for each of the time-varying variables in the dynamical system. These interval annotations tell the compiler the range of values each variable may take on. These annotations can conceivably be automatically derived for some systems using the above techniques.

2.5.7 Scaling

Scaling is used in numerical computation to reduce the effect of numerical error and improve the stability of numerical computations [119, 59]. These numerical scaling transforms are typically highly domain-specific and are often manually derived. Scaling transformations for numerical computations typically only consider the numerical error of the computations.

Practitioners also manipulate the timescale of time-varying dynamical systems comprised of ODEs to render the system more amenable to simulation and linearization [103]. These approaches typically introduce a tunable time scale parameter into

the numerical computation that multiplies all the derivatives by a constant factor.

Relationship to this work: The compiler presented in this thesis automatically scales the quantities in the analog circuit so that the signals within the circuit respect all of the physical limitations of the hardware. The compiler considers a wide range of low-level analog behaviors and physical constraints when scaling the circuit. In contrast, numerical scaling approaches focus on numerical stability and numerical error.

The compiler uses the same basic approach employed by numerical time scaling approaches to change the timescale of the computation. Unlike numerical approaches, the compiler considers analog frequency restrictions imposed by the hardware when changing the timescale and optimizes execution speed instead of numerical stability.

2.6 Conclusion

In this chapter, I presented an overview of the relevant related work for this thesis.

Dynamical Systems: I first provided an overview of dynamical systems. Dynamical systems appear in a wide variety of fields including mathematics, physics, chemistry, biology, economics, engineering, and medicine. Practitioners use dynamical systems to model and predict physical phenomena, implement signal processing and control algorithms on embedded systems, and solve machine learning and optimization problems. A dynamical system typically consists of one or more interdependent variables which change over time. Dynamical systems are typically implemented as a system of ordinary differential equations (ODEs) or partial differential equations (PDEs). The work presented in this thesis focuses on dynamical systems are implemented with ODEs.

Ordinary differential equation solvers simulate systems of ordinary differential equations on digital hardware. Classical differential equation solvers segment time into discrete time steps and compute the system's state at each time step. Digital ODE solvers have difficulty simulating dynamical systems with both fast and slow dynamics accurately and efficiently. The ODE solver must segment time at a fine

granularity to accurately simulate such a dynamical system. For this reason, among others, digital solvers cannot always efficiently simulate non-linear dynamical systems and dynamical systems which operate on different time scales. Other digital simulation approaches which have more predictable performance characteristics primarily work with linear dynamical systems. Practitioners often independently introduce approximations into the dynamical system to make it more amenable to efficient digital simulation.

In this thesis, I simulate dynamical systems with analog hardware. Under this paradigm, the evolution of the currents and voltages within the analog device capture the dynamical system's dynamics over time. This computational model does discretize simulation time and can operate on non-linear dynamical systems.

Historical Analog Computers: I then provide an overview of historical dynamical system-solving analog computers. Historically, practitioners used electrical analog computers to perform dynamical system simulations. These analog computers were programmable and configured with potentiometers, switches, and a patchbay. Researchers proposed a variety of compilation and scaling approaches for this hardware. These automated approaches were designed to work with high-precision functional units which could be manufactured with little process variation. Even after these analog computers disappeared from the computational landscape, analog computation remained an important part of many hardware systems.

Modern Dynamical System-Solving Analog Devices: One prominent line of work focuses on designing ultra-low power reconfigurable analog devices that solve dynamical systems. These analog devices are silicon chips manufactured with conventional fabrication processes and leverage the advances made in analog IC fabrication. These hardware platforms consume very little power, deliver predictable performance characteristics, and are capable of interfacing directly with analog signals. Currently, this hardware is programmed directly at a low level with little or no automation. Historical compilation approaches are ill-suited for these modern analog computers because modern analog hardware offers lower precision functional units sensitive to process variation and noise. In this thesis, I target this class of analog devices and

present a compiler that automatically maps general dynamical systems to a reconfigurable analog device of class. This is the first compiler to automatically map dynamical system computations to reconfigurable analog hardware of this class.

Other Kinds of Reconfigurable Analog Devices: I provided an overview of other kinds of reconfigurable analog devices. For each class of devices, I contrast the programming techniques used to target the device with the compilation techniques presented in this thesis. I primarily discuss mixed-signal spiking neural network accelerators and mixed-signal machine learning accelerators. Many of these mixed-signal accelerators offer functional units which directly correspond to computational operations in the high-level program and do not require the compiler to reason about the low-level device physics of the mixed-signal circuits. In contrast, the work presented in this thesis identifies non-trivial mappings between functional units and mathematical operators and automatically reasons about low-level physical behaviors. I also discuss field-programmable analog arrays. The programming techniques for field-programmable analog arrays are complementary to the techniques presented in this work.

Software Techniques: I conclude the chapter with an overview of related numerical computation and compilation techniques. I discussed why code generation and superoptimization techniques are not well suited for circuit synthesis. I then contrasted the circuit synthesis approach employed in this thesis to deductive synthesis approaches to generalized rewrite-based solvers presented in other work. I next discussed how compilation techniques for coarse-grained reconfigurable architectures and FPGA place-and-route techniques relate to the place and route technique presented in this thesis. I then discussed how interval analysis and scaling approaches from numerical computation related to the compilation techniques presented in this work.

Chapter 3

Dynamical Systems

A dynamical system is a system whose state evolves over time. Dynamical systems appear in a wide variety of fields, including mathematics, physics, chemistry, biology, economics, engineering, and medicine. A dynamical system typically consists of one or more interdependent variables which change over time. Dynamical systems are typically implemented as a system of ordinary differential equations (ODEs) or partial differential equations (PDEs) and are used to solve a variety of different problems:

- **Understanding Physical Phenomena**: Dynamical systems computations in the biological sciences are often used for medical dosage optimization, disease prediction, and understanding biological phenomena. [74, 117]. For example, medical practitioners may use a dynamical system to model the effect of a hormone injection on an individual's hormone levels and then use this information to derive an initial dosage. Practitioners are primarily interested in simulating such systems over a window of time and then inspecting the trajectories of the dynamical system variables or retrieving the steady-state (stable) values of the dynamical system variables.

- **Understanding the Environment**: Dynamical systems are deployed on embedded systems to reconstruct higher-order information from environmental signals in real-time [78, 14]. Examples of such use cases include the prediction of the angle of origin of a sound wave from an array of sensors and the tracking

of the orientation of an object from sensor inputs.

- **Interacting the Environment**: Dynamical systems can be used to control actuators such as motors [33, 9]. Typically these dynamical systems produce driving signals to these actuators to accomplish some goal. This sort of use case is common in fields such as robotics and controls. A dynamical system may be used to stabilize a drone or balance an inverted pendulum, for example. These sensor-actuator applications typically have real-time performance requirements and are sometimes implemented on resource-constrained digital devices such as microcontrollers.

While dynamical systems are invaluable in many different domains, there are challenges with executing these systems accurately and performantly:

- **Stiff Dynamics**: Studying dynamical systems on digital hardware can be challenging as systems are often stiff and therefore prone to numerical instability [94, 36]. In this context, a stiff system is a system that is prone to numerical instability unless the time steps taken are extremely small. For example, many biological systems are stiff systems with both fast-evolving and slow-evolving dynamics. When possible, practitioners will often substitute fast-evolving dynamics with closed-form solutions to resolve stiffness issues with the target dynamical system. This substitution approximates the original behavior because it assumes the fast dynamics reaches a steady-state (stable) solution instantaneously.

  Nonlinear Dynamics: Non-linear dynamical systems are often difficult to simulate as they are more difficult to analyze than linear systems and often use expensive operators. Practitioners often linearize dynamical system non-linearities to make the computations tractable amenable to efficient digital simulation. For dynamical systems which model physical phenomena, this reduces the accuracy of the model in relation to the corresponding physical system [37, 138, 21]. For dynamical systems which observe and interact with the environment, this can reduce the fidelity of the produced outputs.

Real-time Dynamical Systems on Embedded Devices: Dynamical systems which interact with the environment often need to continuously process signals in real-time. Because these applications are typically run on embedded systems, these dynamical system implementations must meet these performance requirements on energy- and compute-constrained platforms. Typically, these dynamical systems are discretized and aggressively simplified so that they can performantly execute on an embedded digital device such as a microcontroller. Even with these simplifications, attaining real-time performance may still be difficult if the sensed values evolve too quickly.

In this chapter, I provide an overview of dynamical systems and describe how these systems are modeled and solved today. I then introduce a dynamical system specification language (DSS) for describing dynamical systems. In Chapter 6, I use the dynamical system specification language described in this chapter to specify twelve benchmark applications from the physics, biology, controls, and robotics domains and two real-time signal processing applications.

3.1 Dynamical System Overview

A dynamical system is a system made up of one or more variables that evolve over time. Dynamical systems are typically modeled with ordinary differential equations (ODEs) or partial differential equations (PDEs). There also exist other, less commonly used, dynamical system formulations, such as integro-differential and delay equations [73].

In this work, I focus on dynamical systems which can be modeled with ordinary differential equations. An ordinary differential equation (ODE) is a differential equation in which all derivatives are taken with respect to an independent variable such as time. ODEs support a narrower class of systems than PDEs as PDEs also allow for partial derivatives to be taken with respect to other dynamical system variables. I present the general formulation of time-varying ordinary differential equations below:

$$F(t, \vec{x}, \vec{x}', ..., \vec{x}^{(n)}) = 0$$

In the above formulation, the time t is the dynamical system time, and the vector $\vec{x}$ is the vector of dynamical system variables. All variables in the above formulation take on real values. The initial state of the dynamical system $\vec{x}(0) = \vec{x}_0$ is the vector of starting values for each variable in the system. Each dynamical system variable starts at its initial state and evolves over time in accordance with the above dynamics. The above formulation is general but more difficult to solve. Typically dynamical system variables are called state variables, and the system's initial state is referred to as the initial condition. In this work, I target a subset of ODEs called explicit ODEs. Explicit ODEs can be rewritten to take on the following form:

$$F(t, \vec{x}, \vec{x}', ..., \vec{x}^{(n-1)}) = x^{(n)}$$

The compiler requires the input dynamical system to be expressed as a set of first-order explicit differential equations. Any system of explicit ODEs can be reduced to an explicit first-order system of ordinary differential equations:

$$\vec{x}' = F(t, \vec{x})$$

The left-hand side of the relation is the first-order derivative of each state variable, and the right-hand side of the relation specifies the dynamics of each state variable over time. Each state variable's dynamics is written as an expression over state variables and time. A system of first-order ODEs is *linear* if each expression is a linear combination of state variables and time. Linear systems of ODEs are desirable because they are easier to implement efficiently and are amenable to a broader set of analyses and alternate representations [58, 98, 99, 17, 77]. Practitioners often linearize non-linear systems to enjoy these benefits. These linearization techniques may reduce the efficacy or accuracy of the dynamical system since it involves approximating the dynamics of the system.

External Inputs: In these systems, external inputs from sensors, for example, are

often written as uninterpreted functions of time ($g(t)$).

3.1.1 Execution of First-Order ODEs

Practitioners are typically interested in *simulating* a dynamical system. Generally, dynamical system simulation techniques compute the trajectories of the state variables over simulation time. I focus on simulation techniques for first-order ODEs:

Digital Simulation: Numerical solvers simulate dynamical systems by discretizing time into small chunks and computing the state of the system at each point in time. It maintains a decimal vector $\mathtt{x}$ of state variable values and the current simulation time. The state vector is instantiated to the initial state of the dynamical system at the start of the simulation. The state is then updated for each time step:

$$\begin{aligned} \mathtt{x} &:= \quad \mathtt{x} + \delta t \cdot F(t + \delta t, \mathtt{x}) \\ \mathtt{t} &:= \quad \mathtt{t} + \delta t \end{aligned}$$

The above relations compute state of the system at $\mathtt{t} + \delta t$ from the state of the system at time $\mathtt{t}$. The time step δt determines how much to advance simulation time. This time step may be fixed or adaptively adjusted from simulation statistics [22, 10]. Any behaviors which occur between time $\mathtt{t}$ and time $\mathtt{t} + \delta t$ are not captured in the digital simulation. Therefore, while taking larger steps results in a faster simulation, it may produce inaccurate signal trajectories.

Analog Simulation: First-order ODEs may also be simulated with programmable analog devices. Under this paradigm, the analog device is itself a dynamical system that evolves over time. First, the analog substrate is configured such that the dynamics of the voltages and currents in the device match the dynamics of the state variables in the target dynamical system. Then the substrate is manipulated so that its initial state matches the initial state of the dynamical system. The dynamical system is then simulated by observing the relevant properties in the analog substrate over time.

Analog devices operate in the continuous time-domain, circumventing the time

scale issues that often plague stiff dynamical system computations implemented with discretized time steps [134]. Time does not tick in intervals as in standard clocked digital systems but instead runs continuously and asynchronously. Because dynamical system time is mapped to hardware time, the time required to simulate the dynamical system can be statically computed before execution. The analog computational model is also massively parallel. While larger dynamical systems use more substrate and therefore consume more area, they do not necessarily take any longer to execute than a smaller dynamical system.

3.1.2 Changing the Speed of First-Order ODEs

The compiler presented in this thesis can tune the execution speed of the dynamical system by leveraging a time-scaling property of first-order ODEs. Given a system of first-order ODEs, the speed at which the trajectories evolve can be manipulated by multiplying the derivatives of all the state variables by a constant coefficient:

$$\vec{x}' = \alpha \cdot F(t, \vec{x})$$

The above equation adjusts the system of first-order ODEs to execute at αx the original simulation speed. Choosing an α value less than one causes the trajectories to evolve more slowly, reducing the simulation speed of the dynamical system. Choosing an α value greater than one causes the trajectories to evolve more quickly, increasing the simulation speed of the dynamical system.

3.2 The Dynamical System Specification Language

This work presents a specification language for dynamical systems. This language is used to specify dynamical systems to the compiler. The dynamical system specification language (DSSL) is the high-level language for this system. The language supports defining dynamical system state variables and functions and annotating variables with interval bounds and frequency limits. The DSSL works with the subset

```
x ∈ RealNumbers, v ∈ Literals
n ∈ NaturalNumbers,
x+ ∈ RealNumbers ≥ 0,
VarList ::= v | VarList,v
I   ::= [x1,x2]
E   ::= E1 + E2 | E1*E2 | x | v
      | integ(E1,E2) | (E) | call(v,EList)
EList ::= E | E,ExprList
F   ::= F1 + F2 | F1*F2 | x | v
      | integ(F1,F2) | (F)  | F1/F2
      | sgn(F) | ln(F)  | exp(F) | sin(F) | cos(F)  | abs(F)
      | min(F1, F2) | max(F1, F2) | pow(F1, F2)
FuncDecl ::= v(VarList) = F
```

Figure 3-1: Math expressions

of mathematical expressions that can be formally described with the expression language presented in Section 3.2.1. The hardware specification languages presented in Chapter 5 also reference the expression language.

3.2.1 Mathematical Expression Language

Figure 3-1 presents the space of mathematical expressions supported by the compiler. These expressions contain both variable literals `v` and real numbers `x`. The expression language offers both a basic expression construct E that defines operators typically natively supported in analog and mixed-signal devices and an expanded expression construct F that extends the basic expression construct to include a wider set of operations. The operations supported in the basic expression construct E are summarized below:

- **Addition and Multiplication**: The addition (E_1 + E_2) and multiplication E_1*E_2 operators implement addition and multiplication over basic expressions, respectively.
- **Integration**: The integration integ(E_1,E_2) operation integrates the derivative E_1 over time. The initival value of the integrated signal is E_2.
- **Function Invocation**: The function invocation operation `call(v,lst(E))`

invokes a user-defined function with the name `v` with a list of of basic expressions `EList` as function inputs. The body of the function is typically defined outside of the expression using a a function declaration statement `v(VarList) = F` which maps a function definition to its implementation. The `v` literal specifies the name of the function and the variable literal list `VarList` is the list of function argument names. The extended expression `F` is the body of the function. The body of the function may only reference variables which are function arguments.

The extended expression construct `F` supports all mathematical operators already supported in `E`.It extends the set of supported operators to include a variety of non-linear mathematical operations:

- **Minimum and Maximum**: The minimum `min(`$\mathtt{F}_1$`, `$\mathtt{F}_2$`)` and maximum `max(`$\mathtt{F}_1$`, `$\mathtt{F}_2$`)` operations take the minimum and maximum of the extended expression operations $\mathtt{F}_1$ and $\mathtt{F}_2$ respectively.

- **Exponentiation**: The exponentiation operation `pow(`$\mathtt{F}_1$`, `$\mathtt{F}_2$`)` raises the expression $\mathtt{F}_1$ to the expression $\mathtt{F}_2$.

- **Natural Log and Natural Exponentiation**: The natural log `ln(F)` and natural exponentiation operations `exp(F)` compute the natural log ($ln(F)$) or the exponential (e^F) of the expression argument `F`.

- **Sine and Cosine**: The sine `sin(F)` and cosine operations `cos(F)` compute the sine and cosine of the expression argument `F`.

- **Absolute Value and Sign**: The absolute value `abs(F)` and sign operations `sgn(F)` take the absolute value and the compute the sign (1 or -1) of the expression `F`. Both of these functions are non-linear and introduce discontinuities into the expression at zero.

```
Stmt ::= var v = E | extern v | interval VarList = I
  | freq VarList = x | func FuncDecl | time x
  | realtime I
Body ::= Stmt | Body ; Stmt
Prog ::= prog v { Body }
```

Figure 3-2: Grammar for DSSL

3.2.2 Dynamical System Specification Language

Figure 3-2 presents the dynamical system specification language (DSSL). The language supports binding symbolic expressions to dynamical system variables and specifying the interval bound and optionally the frequency bound of each variable. The DSSL offers constructs for defining named functions which can later be invoked in the dynamical system relations. The language constructs are summarized below:

- **Variable Definitions**: Each relation declaration statement `var v = E` binds a variable `v` to an basic mathematical expression `E`. These relations together describe the differential equations and straight-line functions that together specify the dynamical system behavior.

- **Function Definitions**: The DSSL supports extended expression operations provided the expression is enclosed in a user-defined function. These functions may be defined in the DSS with function declaration statements (`func FuncDecl` or `func v(VarList) = F`). Each function declaration defines a named function `v` which accepts a list of named arguments `VarList` and implements the extended expression `F`.

- **Interval Annotations**: The DSSL requires variables be annotated with an interval bound. This interval bound `interval VarList = [`x_1`,`x_2`]` indicates that the listed variables `VarList` will not fall below the value x_1 or exceed the value x_2 at any point in time. The compiler uses these interval annotations to ensure the analog signals capture the full range of values for each variable.

- **Execution Time**: The DSSL requires the end user to specify the execution time of the dynamical system in simulation time units. The execution time

statement `time x` indicates the specified dynamical system should be run for `x` units of simulation time. The compiler uses the specified simulation time to determine the execution time of the mapped computation.

The DSSL also supports defining dynamical systems which work with real-time signals. These real-time signals are continuously evolving external inputs which may be taken from sensors or routed from other mixed-signal devices. The DSSL real-time constructs involve real-time frequencies and real-time latencies that are expressed in terms of wall-clock time. The constructs for working with real-time signals are summarized below:

- **External Variable Definitions**: The DSSL supports describing dynamical systems which work with continuously evolving external inputs. The `extern v` declaration defines an external variable named `v`.

- **Maximum Frequency Annotations**: The DSSL requires each external variable to be annotated with a maximum frequency annotation. Each maximum frequency annotation `freq VarList = x` indicates that the listed variables `VarList` will not exceed the maximum frequency `x`. The specified maximum frequency is the maximum real-time frequency of the signal. For example, the maximum frequency of an externally provided signal `mic` carrying a sound wave from a microphone would be the maximum frequency picked up by the microphone (20 kHz). The corresponding DSS statement would be `freq mic = 20000`. The compiler uses these annotations to determine which hardware features can reasonably be used with the defined external signal.

- **Real-time Simulation Speed**: The DSSL supports defining the acceptable range of simulation speeds for real-time applications. The `realtime [x1,x2]` statement indicates the one unit of simulation time must correspond to between x_1^{-1} and x_2^{-1} seconds of wall-clock time. These annotations ensure that the specified dynamical system evolves quickly enough to react to changes in a real-time external signal, and slowly enough to track the slower evolving dynamics

in the external signal. This annotation is typically only provided if an external variable is defined. For example, a dynamical system that processes a sound wave at 20 kHz may need to evolve at two to three times the external signal speed ($40 - 60kHz$) to process the signal efficiently and accurately. For such a system, one unit of dynamical system time would therefore need to map to at least 0.0166-0.025 milliseconds of wall-clock time. The real-time speed annotation for such a system would be `realtime [40000 , 60000]`.

Before compilation, an interval propagation algorithm fills any the missing interval annotations in the dynamical system specification. It then performs a well-formedness check which ensures that all non-external variables have defined behavior and can be bounded. All external variables must have both interval and frequency bounds. The `time` statement describes how much time (in simulation units) to run the computation for.

3.3 Conclusion

Dynamical system computations appear in a broad range of domains, including mathematics, physics, chemistry, biology, economics, engineering, and medicine. Practitioners use dynamical systems to understand physical phenomena, analyze sensed information from the environment, and control actuators such as motors. In this thesis, I focus on dynamical systems implemented with systems of first-order ordinary differential equations (ODEs). These first-order ODEs capture the evolution of one or more variables over time.

Typically, practitioners simulate first-order ODEs with ODE solvers. Digital ODE solvers discretize simulation time and compute the state of the system at each time step. Digital ODE solvers have difficulty efficiently simulating dynamical systems with non-linear dynamics or systems with dynamics that evolve at different time scales. These simulation issues are exacerbated when run on embedded devices that are heavily resource-constrained and must process stimuli in real-time.

In contrast, analog simulation approaches do not discretize time. Instead, simulation time is directly mapped to wall-clock time. With this simulation approach, the amount of time required to execute a dynamical system can be accurately computed ahead of time regardless of the dynamical system's complexity. This property enables programmable analog devices to deliver predictable performance in the presence of complex dynamics.

In this chapter, I present the dynamical system specification language. The compiler presented in this thesis maps dynamical systems written in the dynamical system specification language to the target analog hardware. The dynamical system specification language supports the specification of systems of first-order ordinary differential equations. The core dynamical system specification supports defining differential equations and functions. The dynamical system requires all time-varying variables have interval annotations which bound the range of values the variable may take on.

The dynamical system specification language also provides constructs for defining real-time signals which process external signals. The language offers constructs for defining the range of acceptable execution speeds, in hertz, for the real-time system. The dynamical system language also supports the definition of external signals. Each externally provided signal must also be annotated with a maximum frequency annotation that indicates how fast the external signal evolves and an interval annotation that indicates the dynamic range of the external signal. These constructs enable the compiler to map applications that work with external signals to the analog hardware.

Chapter 4

Dynamical System Applications

This chapter presents the dynamical system specifications for twelve benchmark dynamical systems and two real-time signal processing applications that continuously process external signals. This chapter presents two types of applications:

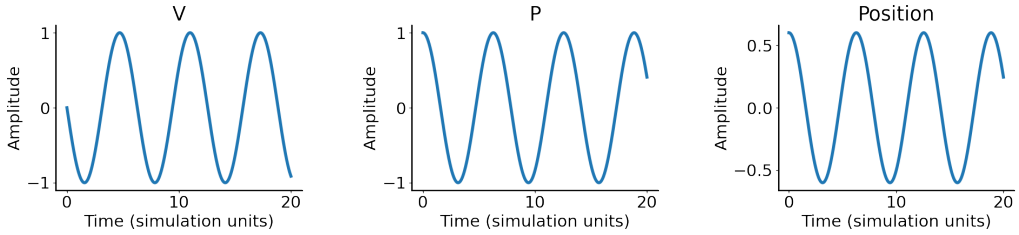
Benchmark Applications: I present twelve benchmark applications, six of which were previously hand-implemented by my collaborators and six of which are novel or from my prior work [4, 2]. For each benchmark application, I describe what the dynamical system application is modeling at a high level and provide a general overview of the dynamical system characteristics. I discuss whether the system is linear or nonlinear and describe each of the variables in the dynamical system. When applicable, I discuss commonly used approximating linearizations that reduce the complexity of the problem, domain-specific variable properties and measures of result fidelity, and analytical techniques for bounding dynamical system variables.

I next present the dynamical system specification describing the dynamical system and provide plots of the dynamical system variable trajectories. I obtain the dynamical system variable trajectories by solving the dynamical system with a high-precision digital solver. I evaluate the compiler on these benchmark dynamical systems in the Chapter 10 of this thesis. The variable trajectories presented in this chapter are the reference trajectories used in Chapter 10.

Real-time Signal Processing Applications: I introduce two real-time signal processing applications that continuously process external signals. These signal process-

ing applications are targeted in the case studies presented in Section 10.10. I discuss the purpose of each signal processing application and provide a breakdown of the associated dynamical system's characteristics. I then present the dynamical system specification for each signal processing application. Note that I do not provide dynamical system variable trajectories for these real-time applications. Because digital dynamical system solvers are designed to simulate closed systems, I cannot use a digital solver to simulate open dynamical systems that work with external signals and interact with the environment.

4.1 Simple Oscillator (`cos`)

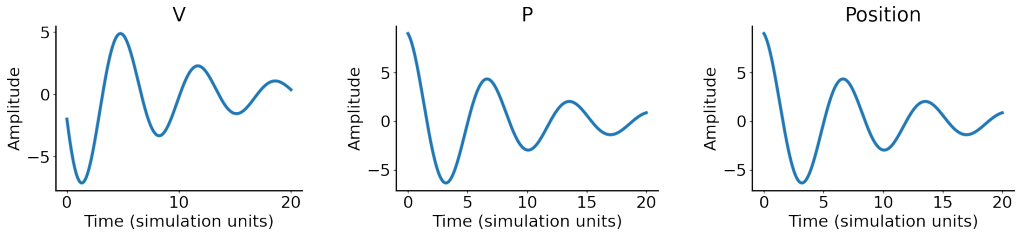



```
var V = integ(-P,0.0);
var P = integ(V, 1.0);
var Position = emit(0.6*P);
interval p = [-1,1];
interval v = [-1,1];
time 20;
```

Figure 4-1: Simple oscillating mass.

The simple oscillator application (Figure 4-1) models the position and velocity of an oscillating mass. The closed-form solutions of the position and velocity of the above system are `cos(t)` and `-sin(t)` respectively. The DSS for this system contains two linear differential equations which model the position `P` and velocity `V` of the oscillator and a straight-line function `Position` that observes the position over time. This system records the position of the mass for 20 units of simulation time.

4.2 Dampened Harmonic Oscillator (cosc)

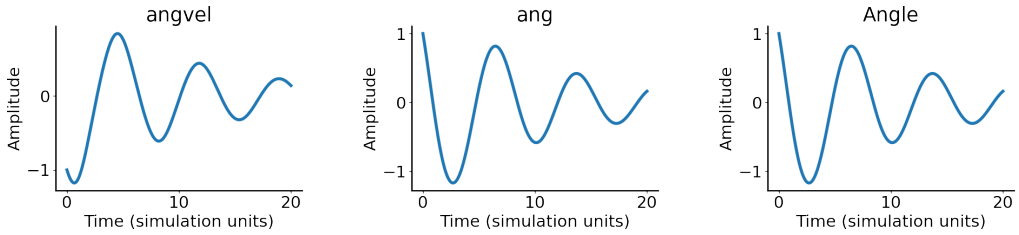



```
var v = integ(-0.22*v - 0.84*P,-2.0);
var p = integ(v, 9.0);
var Position = emit(p);
interval p = [-10,10];
interval v = [-10,15];
time 20;
```

Figure 4-2: Dampened oscillating mass.

The dampened harmonic oscillator application (Figure 4-2) models a mass attached to a spring where the spring exerts some resistance on the mass as it moves. The dynamical system comprises two linear differential equations that model the velocity `v` and position `p` of the mass and one straight-line equation `Position` that records the mass's position over time. The above dynamical system executes for 20 simulation time units. This system is marginally more complex than the simple oscillator as it introduces a `-0.22*v` term that models the effect of the mass's resistance to motion into the equation modeling the velocity of the mass.

4.3 Pendulum (pend)



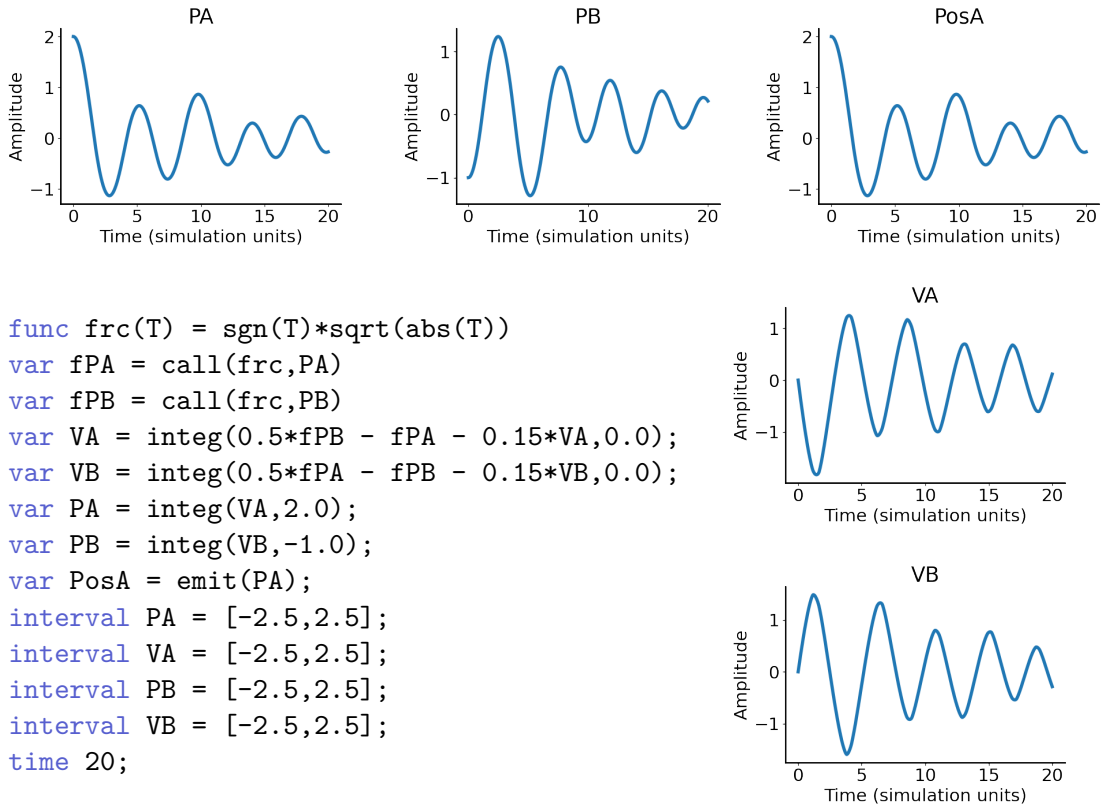

```
func sinf(T) = sin(T)
var angvel = integ(-0.18*angvel-0.8*call(sinf,ang),-1.0);
var ang = integ(angvel, 1.0);
var Angle = emit(ang);
interval ang = [-1.5,1.5];
interval v = [-1.5,1.5];
time 20;
```

Figure 4-3: Movement of a pendulum.

The pendulum application (Figure 4-3) implements a simple physics model which captures the behavior of a swinging pendulum over time. The dynamical system is made up of two differential equations which model the angular velocity `angvel` and angle `ang` of the pendulum and a straight line equation `Angle` which monitors the pendulum angle over time. This dynamical system executes for 20 units of simulation time.

The equation modeling the position is linear, and the equation modeling the velocity is nonlinear in the above system. The pendulum model is a nonlinear dynamical system because it makes use of the `sin` function. The above DSS introduces the `sinf` function which accepts an argument `T` and computes `sin(T)`. Note that a common approximating linearization of the above model involves approximating the sine function with the angle of the pendulum [80]. This approximation is reasonable for situations where the angular velocity Θ' is small.

4.4 Spring (`spring`)



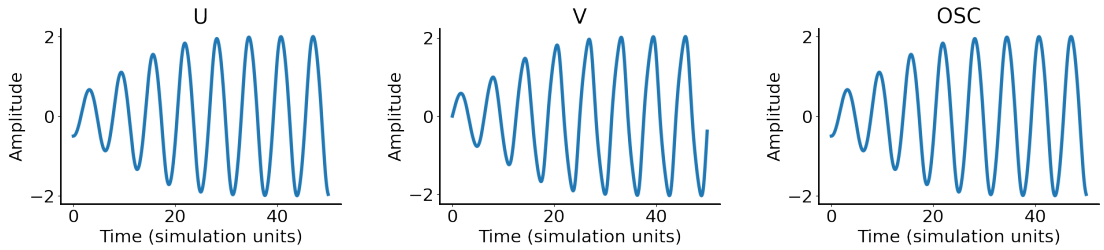

```
func frc(T) = sgn(T)*sqrt(abs(T))
var fPA = call(frc,PA)
var fPB = call(frc,PB)
var VA = integ(0.5*fPB - fPA - 0.15*VA,0.0);
var VB = integ(0.5*fPA - fPB - 0.15*VB,0.0);
var PA = integ(VA,2.0);
var PB = integ(VB,-1.0);
var PosA = emit(PA);
interval PA = [-2.5,2.5];
interval VA = [-2.5,2.5];
interval PB = [-2.5,2.5];
interval VB = [-2.5,2.5];
time 20;
```

Figure 4-4: Two-spring system

The spring application is a physics model which captures the dynamics of two masses `A` and `B` that are linked together by a system of springs. The dynamical system models the position and velocity of masses A and B and records the position of mass A as an external signal. The spring application records the position of the spring for 20 simulation time units.

The equations modeling the velocity of both masses are non-linear as they include `sgn(PA)*sqrt(PA)` and `sgn(PB)*sqrt(PB)` terms respectively. The above DSS writes both of these non-linear terms to the temporary variables `fPA` and `fPB` to reduce the complexity of the velocity expressions. A common approximating linearization for the above non-linear system involves replacing the $sqn(PA)\cdot\sqrt{PA}$ and $sgn(PB)\cdot\sqrt{PB}$ terms with the variables PA and PB.

4.5 Vanderpol Oscillator (`vanderpol`)




```
var V = integ(0.2*(V*(1.0-U*U)) - U,-0.5);
var U = integ(V,0.0);
var OSC = emit(U);
interval U = [-2.5,2.5];
interval V = [-2.5,2.5];
time 10;
```

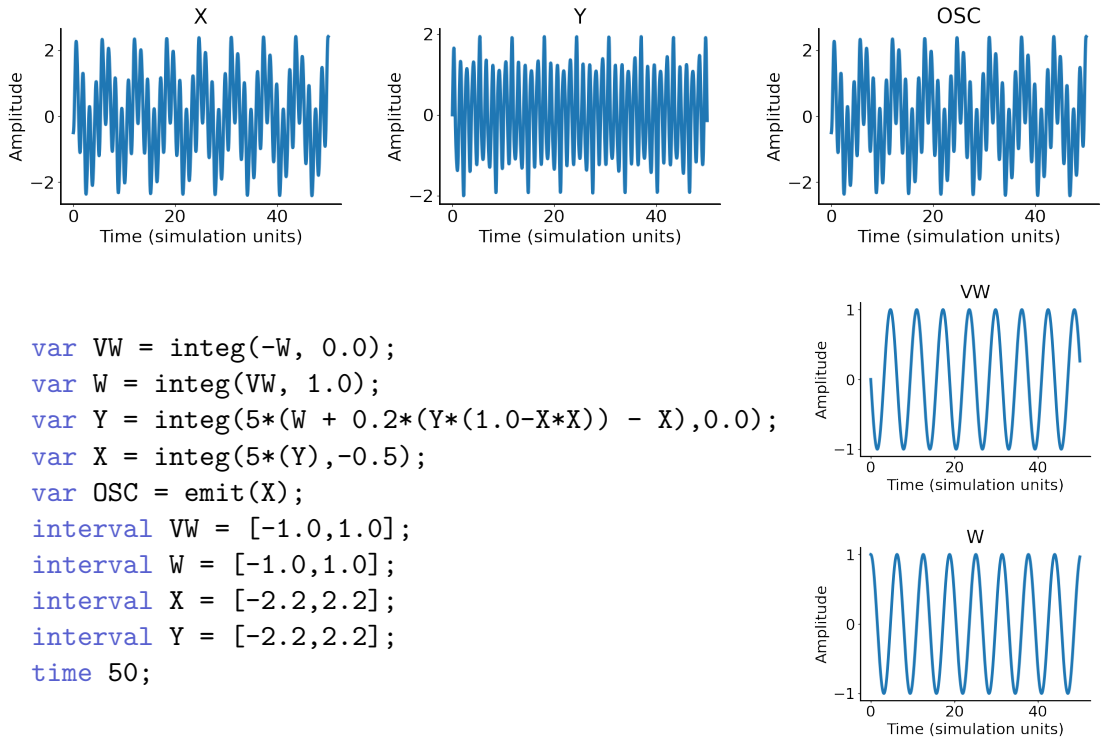
Figure 4-5: Van der Pol oscillator.

The `vanderpol` application (Figure 4-5) executes a two-dimensional non-linear Van der Pol oscillator for 10 units of simulation time. The Van der Pol oscillator model has long been used in the physical and biological sciences to model phenomena ranging from seismic activity to vocal folds. The basic formulation of the Van der Pol oscillator is as follows:

$$\dot{u} = v \qquad \dot{y} = \mu(1 - x^2)y - x + g(t)$$

The `vanderpol` application is a nonlinear dynamical system made up of two differential equations and one straight-line function. This implementation of the Van der Pol oscillator models two variables `V` and `U`. This model is unforced, meaning the external input term $g(t)$ in the above differential equation modeling `V` is set to zero. This application observes the evolution of the oscillating variable `U` over time. The μ parameter in the above implementation is 0.2 and the `V` and `U` variables are initially set to `-0.5` and `0.0` respectively. The bounds for the oscillator variables are elicited by executing the above dynamical system.

4.6 Forced Vanderpol Oscillator (`forced`)



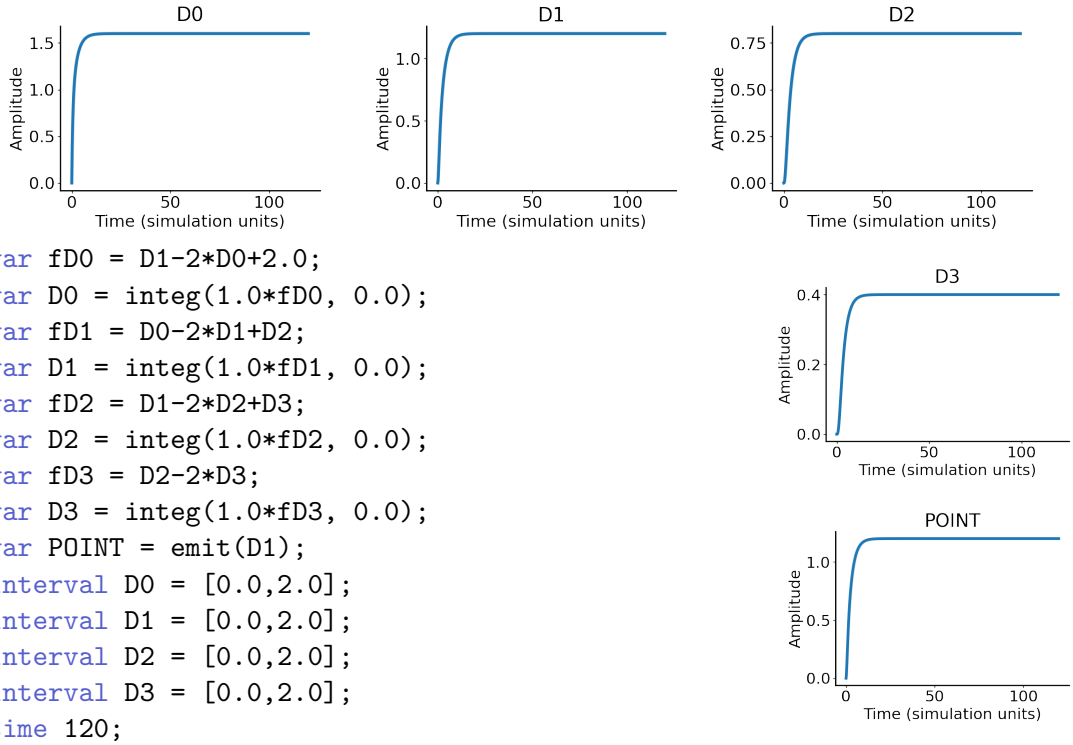

```
var VW = integ(-W, 0.0);
var W = integ(VW, 1.0);
var Y = integ(5*(W + 0.2*(Y*(1.0-X*X)) - X),0.0);
var X = integ(5*(Y),-0.5);
var OSC = emit(X);
interval VW = [-1.0,1.0];
interval W = [-1.0,1.0];
interval X = [-2.2,2.2];
interval Y = [-2.2,2.2];
time 50;
```

Figure 4-6: Van der Pol oscillator with an oscillating forcing function.

The `forced` application (Figure 4-6) extends the basic Van der Pol oscillator from Section 4-5 to accept a forcing function. This system is nonlinear and contains four differential equations and one straight-line function. The forced Van der Pol oscillator internally generates an oscillating function with a velocity `VW` and position `W` that implements the closed form function `cos(t)`. The position `VW` of the oscillating input is provided as the forcing function `g(t)` to the Van der Pol oscillator.

In the above implementation, the Van der Pol oscillator differential equations modeling `Y` and `X` are each scaled by a factor of five. This scaling coefficient modifies the oscillator to evolve 5x faster than provided the forcing function. The bounds for the oscillator variables are elicited by executing the above dynamical system.

4.7 1D Heat Model (heatN4X2)



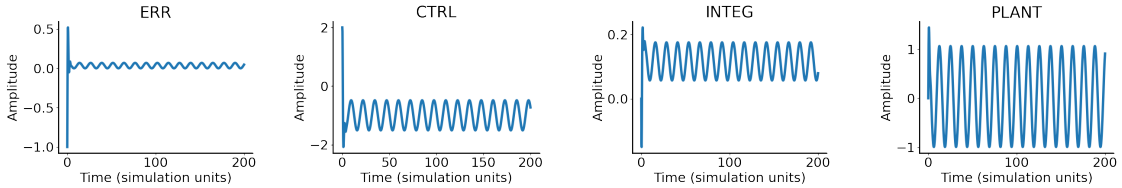

```
var fD0 = D1-2*D0+2.0;
var D0 = integ(1.0*fD0, 0.0);
var fD1 = D0-2*D1+D2;
var D1 = integ(1.0*fD1, 0.0);
var fD2 = D1-2*D2+D3;
var D2 = integ(1.0*fD2, 0.0);
var fD3 = D2-2*D3;
var D3 = integ(1.0*fD3, 0.0);
var POINT = emit(D1);
interval D0 = [0.0,2.0];
interval D1 = [0.0,2.0];
interval D2 = [0.0,2.0];
interval D3 = [0.0,2.0];
time 120;
```

Figure 4-7: One-dimensional heat model with four points.

The `heat` application implements a one-dimensional, grid-based model of the heat equation PDE. This application models the heat moving through a one-dimensional line of points. This system is a linear system comprised of four differential equations which each model one point in the line. This system observes the heat at the second point (`D1`) for 120 units of simulation time.

In the above equation, the `D0`, `D1`, `D2`, and `D3` variables correspond to the first, second, third and fourth points in the line. each point both accepts heat from and releases heat to the neighboring points. The first point `D0` is consistently supplied with two units of heat. The heat then flows from the first point to all the other points. All of the variables are bounded by using the principle of conservation of heat – no single point can have more heat than the total amount of heat put into the system. The intermediate variables `fD0`, `fD1`, `fD2`, and `fD3` store the derivatives of the state variables `D0`, `D1`, `D2`, and `D3`. The DSS defines these intermediate variables to ensure the compiler processes the derivative expressions separately.

4.8 PID Controller (pid)




```
var VSIG = integ(-0.25*VSIG, 0.0);
var SIG = integ(VSIG, 1.0);
interval VSIG = [-0.5,0.5];
interval SIG = [-1.0,1.0];

var ERR = PLANT-SIG;
var CTRL =-2.0*ERR-8.0*INTEG;
var INTEG = integ(ERR - 0.3*INTEG,0.0);
var PLANT = integ(CTRL + 1, 0.0)
var Controlled = emit(CTRL);
interval PLANT = [-2.0,2.0];
interval CTRL = [-2.0,2.0];
interval ERR = [-2.0,2.0];
interval INTEG = [-2.0,2.0];
time 200;
```

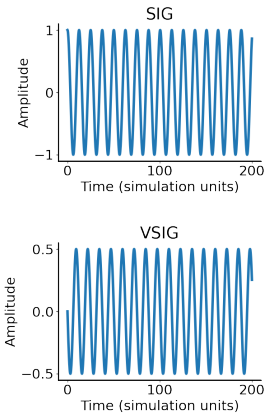


Figure 4-8: Proportional-integral controller.

The `pid` application is a proportional-integral controller that computes a compensating signal that attenuates away the unwanted dynamics from an oscillating input. The above application is a linear system made up of three straight line equations and four differential equations and is executed for 200 units of simulation time.

The `pid` application internally generates an oscillating signal with velocity `VSIG` and position `SIG` that implements the closed-form function `cos(0.25*t)`. The oscillator position `SIG` is the reference function the PI controller is trying to match. The `PLANT` signal is the output of an unknown system that accepts an input signal. The goal of the PI controller is to find the control signal which causes the output of the unknown system `PLANT` to match the reference signal `SIG`. I use a simple unknown system that integrates the provided control signal and adds a fixed error of `1` to the generated output. The `ERR` variable tracks the error between the observed and reference signal.

The PI controller generates a control (`CTRL`) signal that is provided as an input into an unknown system. The control signal contains both a term that is proportional to the observed error (`-2.0*ERR`) and the integrated error over time (`-8.0*INTEG`). The integrated error `INTEG` starts at zero and computes the integral of the error signal over time with some leakage.

In the above figures, the error of the system `ERR` quickly converges to approximately zero. After some initial fluctuations, the reference signal `SIG` and observed signal `PLANT` follow the same trajectory.

4.9 Kalman Filter (kalman)

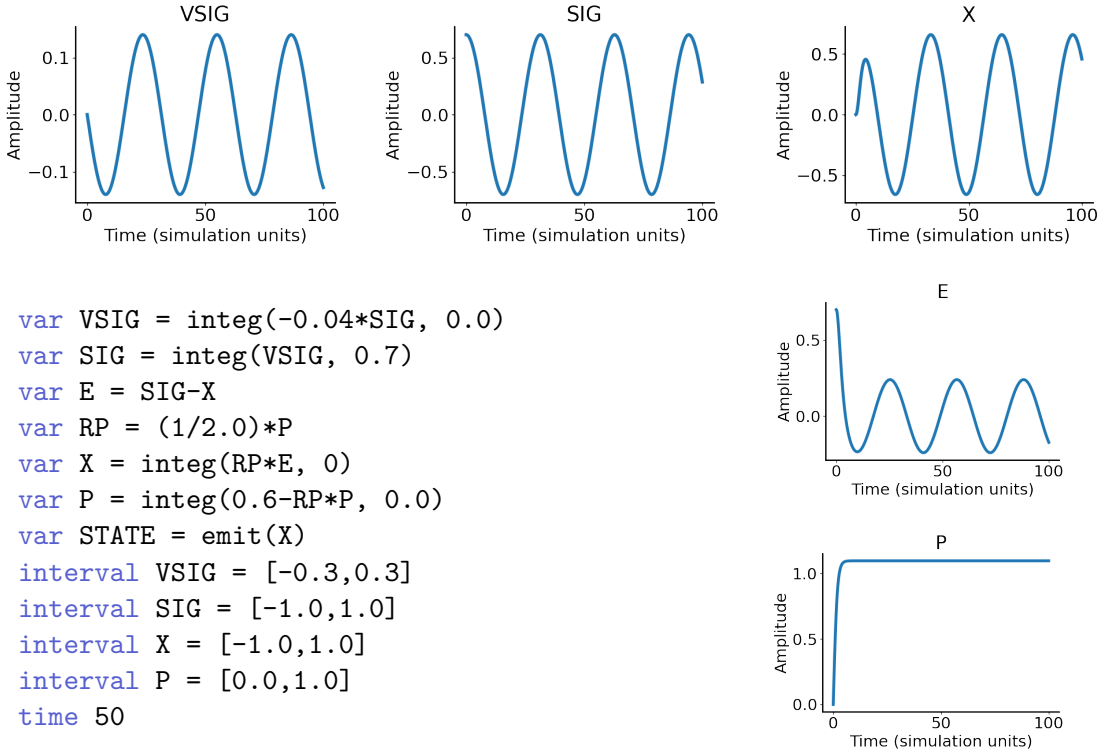


Figure 4-9: Noise-eliminating kalman filter.

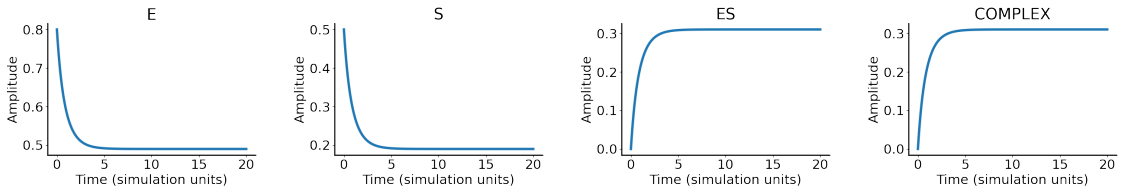
The continuous-time Kalman filter tracks the average of a noisy signal for 50 simulation units [78]. Typically, Kalman filters are implemented discretely using an algorithm which predicts the next state of the system and an update step which studies the agreement between the prediction and measured state and updates the underlying model accordingly. This continuous-time formulation performs both steps simultaneously and continuously. The model used by the Kalman filter tracks the estimated state of the signal and the accuracy of the state estimation. The general model of the one-dimensional continuous-time continuous-observation average tracking kalman filter is as follows:

$$\begin{aligned} Error &= Input - State \\ State &= \int ProcNoise^{-1} \cdot Cov \cdot Error & State(0) &= State_0 \\ Cov &= \int MeasNoise + ProcNoise^{-1} \cdot Cov^2 & Cov(0) &= 0.0 \end{aligned}$$

The *State* variable tracks the predicted noiseless trajectory of the input signal over time, and the *Cov* variable tracks the process noise variance over time. The process noise captures the error in the prediction that results from modeling approximations. The *State* variable is continuously updated with the measured error between the observed input and the prediction. The initial value of the *State* variable is typically an initial guess. The *Cov* variable is also continuously updated to account for measurement noise and noise arising from uncaptured dynamics. The $MeasNoise$ and $ProcNoise$ parameters specify the degree of measurement and process noise present in the system.

The above `kalman` application implements a Kalman filter which denoises an input signal. It implements a nonlinear system made up of six differential equations and three straight line functions. The `VSIG` and `SIG` variables model the position and velocity of a oscillator which implements the closed-form function `0.7*cos(0.2*t)`. This signal `SIG` is used as an input to the Kalman filter. The Kalman filter then tracks the provided signal using the above model with a measurement noise of 0.1 and a process noise of 2.0. The variable `X` is the state of the system, and the variable `P` is the estimated variance of the process noise. Note that the quantity $ProcNoise^{-1} \cdot Cov$ is stored in the temporary variable `RP`. This optimization improves the agreement between the `0.5*P*P` and `0.5*P` terms in the `X` and `P` differential equations respectively. The interval bounds for all the variables are derived by exercising the above dynamical system.

4.10 Michaelis Menten Reaction (smmrxn)




```
var E = 0.8-ES
var S = 0.5-ES
var ES = integ(E*S - 0.3*ES, 0.0)
var COMPLEX = emit(ES)
interval E = [0.0,0.8]
interval S = [0.0,0.5]
interval ES = [0.0,0.5]
time 20
```

Figure 4-10: Michaelis-Menten chemical reaction.

The Michaelis-Menten chemical reaction is a simple chemical reaction which models the formation of an enzyme-substrate complex from two reagents [95]. The Michaelis-Menten chemical reaction can be summarized with the following reaction equations:

$$E + S \rightarrow_{k_f} ES$$
$$ES \rightarrow_{k_r} E + S$$

In the first reaction equation, the enzyme E and substrate S come together to form the enzyme-substrate complex ES. The formation rate of this compound is k_f. The second reaction equation models the decomposition of the enzyme-substrate complex back into the enzyme and substrate reagents. The dissassociation rate of this compound is k_r. These rates are typically empirically measured through wet-lab experiments. The following set of differential equations models the above system:

$$\dot{E} = k_r \cdot ES - k_f \cdot E \cdot S \qquad E(0) = E_0$$
$$\dot{S} = k_r \cdot ES - k_f \cdot E \cdot S \qquad S(0) = S_0$$
$$\dot{ES} = k_f \cdot E \cdot S - k_r \cdot ES \qquad ES(0) = ES_0$$

In the above model, the reagents E and S are depleted at the same rate $k_f \cdot E \cdot S$ the complex ES is formed. The complex ES is depleted at the same rate $k_r \cdot ES$

the enzyme E and substrate S are formed. Both of these rates are influenced by the reaction rate constants defined above. This system can be futher simplified by leveraging the fact that these reactions are occuring in a closed system. In this closed system, the following conservation equalities must hold:

$$E + ES = ES_0 + E_0$$
$$S + ES = ES_0 + S_0$$

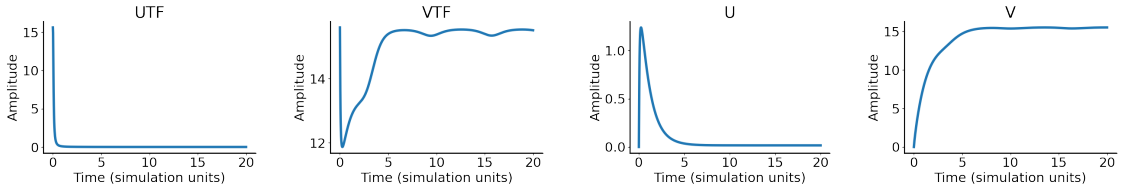
The first equality specifies that the total amount of enzyme and complex in the system must equal the initial amount of enzyme and complex in the system. Because no enzyme can be added or removed from the system at any time, all the available enzyme was either initially in the system or part of the starting amount of the complex ES. The same applies to the substrate S. I then use these conservation relations to simplify the system:

$$E = E_0 + ES_0 - ES$$
$$S = S_0 + ES_0 - ES$$
$$\dot{ES} = k_f \cdot E \cdot S - k_r \cdot ES \quad ES(0) = ES_0$$

The above simplification replaces the differential equations modeling E and S with straight-line equations derived from the above conservation relation.

The `smmrxn` application executes the above simplified dynamical system that models the Michaelis-Menten chemical reaction. The system is a nonlinear system made up of three straight-line equations and one differential equation. This application tracks the concentration of the enzyme-substrate complex `ES` for 20 simulation time units. The formation and disassociation parameters are instantiated to 1.0 and 0.3, respectively. The initial enzyme, substrate, and complex quantities are 0.8, 0.5, and 0.0, respectively. The interval bounds for all the variables are derived from the above conservation equations. The maximum amount of ES in the system is the initial amount of the limiting reagent $min(E_0, S_0)$. All variables are positive since they are tracking physical quantities (chemical concentrations).

4.11 Genetic Toggle Switch (gentog)




```
var VPERT = integ(-PERT, 0.0)
var PERT = integ(VPERT, 0.5)
var IPTG = 0.5+PERT

func utf(T) = 15.62/pow((1+max(T,0)),2.5)
func vtf(T) = 15.6/(1+max(T,0))
func umod(T) = pow((1+max(T,0)),-2.0015)


var UMOD = call(umod,IPTG)
var UTF = call(utf,V)
var VTF = call(vtf,(U*UMOD))
interval UMOD = [0.0,1.0]
interval UTF = [0.0,16.0]
interval VTF = [0.0,16.0]

var V = integ(VTF - V, 0.0)
var U = integ(UTF - U, 0.0)
var compV = emit(V)

interval V = [0.0,16.0]
interval U = [0.0,1.2]
time 20
```

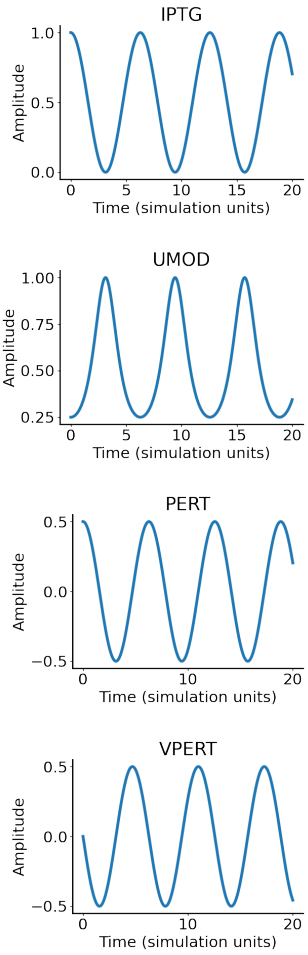


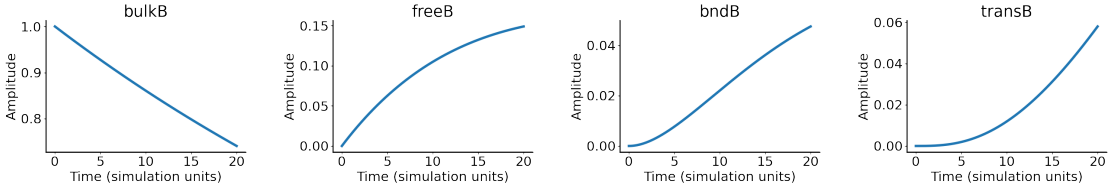
Figure 4-11: Genetic toggle switch.

The genetic toggle switch is a synthetic bi-stable gene regulatory network [46]. It models the activity of the repressor proteins U and V in the presence of a small excitatory molecule $IPTG$. The repressor V binds to U, inhibiting its production. The repressor U also binds to V, inhibiting its production. The small molecule $IPTG$ binds to the repressor U, inhibiting its activity.

The `gentog` application implements the genetic toggle switch. It is a non-linear

dynamical system made up of 2 differential equations and 4 straight-line functions. The `UTF` and `VTF` variables track the transcription rate of the genes which encode the `U` and `V` proteins. The `UMOD` variable captures the interaction between the `IPTG` molecule and the repressor protein `U`.

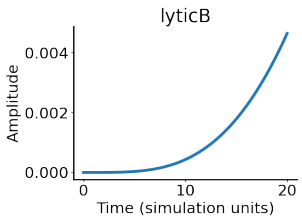
4.12 Botulism Neurotoxin (bont4)




```
var bulkB = integ(-0.015*bulkB, 1.0)
var freeB = integ(0.15*bulkB-0.058*freeB, 1.0)
var bndB = integ(0.058*freeB - 0.141*bndB, 0.0)
var transB = integ(0.141*bndB - 0.013*transB, 0.0)
var lyticB = integ(0.013*transB, 0.0)
var MTRANSB = emit(transB)

interval bulkB = [-1.0,1.0]
interval freeB = [-0.02,0.02]
interval bndB = [-0.03,0.03]
interval transB = [-0.06,0.06]
interval lyticB = [-0.004,0.004]

time 20
```



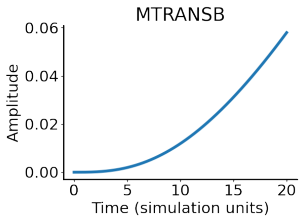


Figure 4-12: Botulism neurotoxin model.

The bont4 application implements a biological model of the botulism neurotoxin pathway [76]. This dynamical system is a linear system with five differential equations and one straight-line function. The following four-stage reaction summarizes the lifecycle of the botulism neurotoxin:

$$bulk \rightarrow^{k_S} free \rightarrow^{k_B} bound \rightarrow^{k_T} translocated \rightarrow^{k_L} lytic$$

First, bulk neurotoxin (*bulk*) is translated to free neurotoxin. Then, the free neurotoxin (*free*) binds to sites on synaptic termini. The bound neurotoxin (*bound*) is then translocated into the neuroplasm (*translocated*). Once the neurotoxin is in the neuroplasm, it exerts its toxic effects (*lytic*). I am primarily interested in observing the level of translocated neurotoxin over time for a system with a starting amount of 1.0 units of bulk neurotoxin. The amount of translocated neurotoxin is observed for 20 simulation time units. All of the rate constants k_S, k_B, k_T and k_L are empirically

measured from wet-lab experiments. The `bont` application above instantiates k_S, k_B, k_T and k_L to `0.015`, `0.058`, `0.141`, and `0.013` respectively. All intervals are derived by executing the dynamical system and observing the trajectories of the quantities.

4.13 Example Real-Time Dynamical Systems

I introduce two real-time applications which operate on externally provided real-time signals.

4.13.1 Bias Shift Detector

```
extern U;
interval U = [-1.0,1.0]; freq U = 40000;
var X = extin(U)
var Y = integ(0.3*X - 0.8*Y, 0.0);
var Trigger = emit(Y);
interval Y = [-1.0,1.0];
realtime [38000,42000];
time 0.0008;
```

Figure 4-13: Bias change detector

The bias shift detector identifies shifts in the average value of an externally provided input. The bias shift detector is designed to be used in conjunction with a thresholding circuit – these two components together can produce a smart interrupt that wakes up a co-processor when the average of the measured signal moves outside of its expected range.

The detector accepts an external input signal `U` with a frequency of `40` kHz and a dynamic range of [-1,1]. The detector produces, as output, a trigger signal (`Trigger`) which tracks the average of the input signal. The bias shift detector must execute at around the same speed as the input signal. The DSS limits the realtime integration speed of the system to 38-42 kHz (`realtime` statement). This frequency restriction

ensures the dynamical system evolves at approximately the same speed as the external signal. This benchmark processes the external signal for a total of 8 milliseconds.

The bias shift detector implements a leaky integrator. A leaky integrator is a linear dynamical system composed of one state variable which integrates a forcing function `X` over time:

$$X' = Y - \alpha \cdot X$$

The α term determines the rate at which the leaky integrator forgets the system's current state. The `Y` term is the external signal to integrate. A leaky integrator is equivalent to a first-order low pass filter with the following transfer function $H(s) = \frac{Y(s)}{X(s)}$. I derive the transfer function below for this system below:

$$\begin{aligned}
\mathrm{L}(Y(t)) = Y(s) = &\quad \mathrm{L}(\int \beta \cdot X(t) - \alpha \cdot Y(t)) \\
&\quad s^{-1}\mathrm{L}(\beta \cdot X(t) - \alpha \cdot Y(t)) \\
Y(s) = &\quad s^{-1}(\beta \cdot X(s) - \alpha \cdot Y(s)) \\
Y(s) + \alpha \cdot s^{-1}Y(s) = &\quad \beta \cdot s^{-1}X(s) \\
Y(s) \cdot (1 + \alpha \cdot s^{-1}) = &\quad \beta \cdot s^{-1}X(s) \\
\frac{Y(s)}{X(s)} = &\quad \beta \cdot s^{-1} \cdot (1 + \alpha \cdot \tfrac{1}{s})^{-1} \\
\frac{Y(s)}{X(s)} = &\quad \beta \cdot (s \cdot \alpha^{-1} + 1)^{-1}
\end{aligned}$$

The above transfer function implements a low-pass filter with a gain of β and a cutoff frequency of α. For the above bias shift detector, the gain of the filter is `0.3`, and the cutoff frequency is 0.8x the baseline integration speed of the dynamical system. The cutoff frequency is therefore between 30.4-33.6 kHz.

4.13.2 Denoiser

```
extern SIG;
interval SIG = [-1.0,1.0];
freq SIG = 40000;

var U = extin(SIG);
var E = U-X;
var RP = 1.0*P;
var X = integ(RP*E, 0.0);
var P = integ(1.0 - RP*P, 0.0);
var Output = emit(X);

interval X = [-1.0,1.0];
interval P = [0.0,1.0];

realtime [38000,42000];
time 0.0008;
```

Figure 4-14: Signal denoiser

The denoiser removes the noise from an externally provided input signal and emits the smoothed signal as output. The denoiser is an example of a real-time analog signal processing application. Such computations can continuously process signals with high-frequency components without requiring signal sampling. Typically, the produced computation result is a lower frequency signal that can be efficiently sampled and processed by a low-power digital processor. Other examples of real-time signal processing applications include frequency de-modulation and generalized state estimation [106, 135].

The detector accepts an external input signal `U` with a frequency of `40` kHz and a dynamic range of [-1,1]. The detector produces, as output, a trigger signal (`Trigger`) which changes level when the average of the input signal changes. The bias shift

detector must execute at around the same speed as the input signal. The DSS limits the realtime integration speed of the system to 38-42 kHz (`realtime` statement). This execution speed restriction ensures that the state estimation computation keeps pace with the externally provided signal. This benchmark processes the external signal for a total of 8 milliseconds.

The denoiser implements the Kalman filter introduced in Section 4.9. The Kalman filter is instantiated with a measurement noise parameter of 1.0 and a process noise parameter of 1.0. The `E`, `X`, and `P` variables implement the error, internal state, and predicted error of the Kalman filter.

4.14 Conclusion

This chapter presents twelve benchmark dynamical systems from the biological, physics, and control systems domains. For each benchmark application, I discuss the approximations typically applied for each type of application and outline any approximations I apply to the dynamical system in the dynamical system specification. When applicable, I discuss how the interval bounds for the dynamical systems could be analytically derived. I also present two dynamical system applications which process real-time signals in this chapter. Each of these real-time applications performs computation on a continuously evolving, externally provided signal.

Further Reading: Chapter 6 presents the analog hardware implementations of the twelve benchmark dynamical systems presented in this chapter on the HCDCv2 reconfigurable analog device. Chapter 10 presents the results from executing the twelve benchmark dynamical systems on the HCDCv2 reconfigurable analog device. The two real-time dynamical system computations presented in this chapter are implemented and evaluated on the target hardware platform in Section 10.10.

Chapter 5

Reconfigurable Analog Devices

The compilation techniques presented in Chapters 7, 8, and 9 automate the process of deriving an analog circuit which implements a given dynamical system computation on a differential equation-solving analog device. The compiler presented in this work [3] targets the HCDCv2 analog device [132, 61, 51]. The HCDCv2 reconfigurable analog device is an ultra-low power differential equation-solving analog device designed to solve non-linear ordinary differential equations. My earlier work on compilation [4, 2] targeted reconfigurable dynamical system-solving analog computing platforms which simulate biological systems [105, 128, 140, 141].

This chapter provides an overview of reconfigurable differential-equation solving analog devices, presents the relevant analog device specification and programming languages utilized by the compiler, and HCDCv2 analog device specification and runtime system.

The target class of reconfigurable analog devices leverages the physical behavior of transistors to implement computation. Under this paradigm, voltages and currents within the device implement continuous variables in the computation. These devices are programmed by routing together configurable analog blocks using programmable interconnects to form a circuit. To faithfully implement a computation, the physics of the circuit must capture the underlying dynamics of the dynamical system so that the trajectory of each variable can be recovered at runtime by applying a recovery transformation. This class of differential equation-solving analog devices are typically

programmed in two ways:

- **Connection Formation**: Connections between blocks are formed by digitally enabling interconnects on the device.

- **Block Configuration**: Each block in the analog device is digitally configurable. Blocks are typically parametric and may be digitally programmed to implement one of many different functions. Some blocks may also accept digitally settable data fields which appear in the input-output relation implemented by the block.

Together, these two configuration techniques are used to implement circuits comprised of configured blocks. Once the circuit is programmed into the analog device, the dynamical system is executed by powering on the circuit and observing the currents and voltages over time. Each of these currents and voltages models a variable or expression from the dynamical system. The compiler is responsible for constructing the analog circuit that captures the dynamics of the target dynamical system. compilation problem is challenging for several reasons:

- **Complex, Non-Standard Blocks**: There is no universally agreed-upon collection of analog blocks – the devices instead provide highly specialized blocks that implement anything from simple functions to sets of differential equations. These blocks are highly configurable and often can be reconfigured to implement a multitude of different functions. The compiler may need to identify non-trivial compositions of these blocks to implement the desired functions.

  In analog devices, not all computation is performed with analog blocks, and not all blocks perform computation. Analog devices which work with analog currents do not provide adder blocks to the compiler and typically require addition operations to be implemented with Kirchhoff's law. These same analog devices provide copier blocks to duplicate analog currents. These blocks do not perform computation but must be used to route signals to multiple places.

- **Restrictive Routing Environment**: Analog devices typically provide highly restrictive interconnects and offer few digitally programmable connections. The compiler, therefore, must carefully map all the blocks in the derived circuit to locations on the device. The compiler may need to insert additional blocks to implement connections between blocks that are far apart.

- **Low-Level Physical Behaviors**: Analog devices are subject to analog noise and process variation and exhibit value- and frequency-dependent non-idealities. While the hardware designer does mitigate some of these behaviors in the device design, the compiler must automatically compensate for all remaining physical behaviors in the derived circuit. One common mitigation strategy includes changing the dynamical system parameters written to the circuit. This process is called scaling the circuit.

The compilation techniques presented in Chapters 7, 8, and 9 automatically address the above challenges. This chapter covers the following topics:

- **Low-Level Programming Interfaces of Analog Devices**(Section 5.1): I describe the kinds of programmable degrees of freedom included in analog designs. I then describe the high-level abstractions I use to capture these low-level programming constructs. These high-level abstractions make up the device programming interface.

- **Low-Level Physics and Analog Devices**(Section 5.2): I describe the low-level physical phenomena which introduce unwanted behaviors in the analog device. I discuss which physical phenomena are mitigated at an architectural level and how each mitigation strategy works. For the phenomena which are mitigated at an architectural level, I describe how to mitigate each unwanted physical behavior. I then describe how the compiler automatically handles these unwanted behaviors at a higher level of abstraction.

- **Delta Models and Calibration**(Section 5.3): I introduce the concept of a delta model. Delta models are used to capture process variation-induced behav-

ioral variations in the hardware. The compiler uses the delta model information to more accurately target the device on hand. I describe what delta models look like, how the compiler uses the delta models, and how they are elicited from the device on hand.

- **Analog Device Specification Language (ADSL)**(Section 5.5): I present the analog device specification language. The ADSL describes the physical limitations and behaviors present in the analog device. The ADSL also specifies the programming interface and high-level behavior of the target dynamical system-solving analog device.

  Section 5.5.1 describes how programmable functional blocks are specified in this language. This section also describes how the block specifications encode the block operating range and frequency limitations, block noise, block quantization error, and block delta models. Section 5.5.2 describes the language constructs offered to specify the device layout and the programmable connections available in hardware. The analog device specification language grammars make use of the grammar shortcuts presented in Section 5.4 of this chapter.

- **Analog Device Programming Language**(Section 5.6): I present the high-level analog device programming language (ADPL) that is used to program the reconfigurable analog device. The analog device programming language configures analog blocks and links block ports together with digitally settable connections. The analog device programming language grammars make use of the grammar shortcuts presented in Section 5.4.

- **HCDCv2 Analog Device Specification**(Section 5.7): I present the analog device specification (ADS) for the HCDCv2 analog device. This section provides a detailed overview of each block's high- and low-level programming interfaces and physical limitations. Section 5.7.1 presents the specification for the programmable blocks in the device. Section 5.7.2 presents the programmable connections available on the HCDCv2 device and the overall device layout.

- **HCDCv2 Calibration and Delta Models**(Section 5.8): I discuss the calibration strategies implemented by the HCDCv2. I present a multiplier case study where I walk through how the delta models are elicited from the hardware and used by the compiler. I discuss the effect of the calibration strategy on the compiler's ability to target the device. The HCDCv2 supports two software-defined calibration algorithms which are implemented in the device firmware. The firmware implements a traditional (`minimize_error`) calibration algorithm that aims to eliminate all unexpected behavior and a co-designed calibration algorithm (`maximize_fit`) that prioritizes eliminating behavioral deviations which cannot be compensated for in compilation. I provide an overview of how the calibration algorithms affect the inferred block delta models and provide a multiplier case study that concretely illustrates the interplay between the calibration algorithm and the elicited delta model for a fabricated multiplier instance.

- **HCDCv2 Software Stack and Runtime**(Section 5.9): I discuss the HCDCv2 low-level programming interface and the calibration, characterization, and circuit execution procedures for the target HCDCv2 analog device. This discussion provides some insight into the operation of the hardware underneath the provided hardware abstraction.

  Section 5.9.1 presents the HCDCv2 low-level programming interface used to write circuits to the HCDCv2, calibrate the HCDCv2, and characterize blocks on the HCDCv2. Section 5.9.2 presents an overview of the databases used to store the device-specific calibration and characterization information. Section 5.9.3 describes the HCDCv2 runtime procedures used to populate the calibration and characterization databases – these runtime procedures are performed offline before the device is used to execute dynamical systems. Section 5.9.4 describes the workflow for executing an ADP on the HCDCv2.

5.1 Programmability of of Analog Devices

Reconfigurable analog devices offer blocks that may be routed together by enabling digitally programmable interconnects. Each block on the analog device is digitally reconfigurable and may be programmed to implement a variety of different functions. Internally, each block is a mixed-signal circuit comprised of circuit components such as capacitors, resistors, and transistors. The designer typically makes the circuit programmable in several different ways:

- **Changing the Circuit Structure**: The designer may introduce digitally settable bits which redirect how signals are routed within the analog circuit. This design feature enables the hardware designer to implement multiple functions with the same circuit.

- **Current and Voltage Sources**: The designer may introduce programmable current and voltage sources into their design. Each of these current and voltage sources accepts a bit vector and translates the bit vector to a current level. The encoding scheme used by the source varies depending on the underlying design.

- **Digital Logic**: The designer may introduce digital logic into their design to improve the expressivity of the device. For example, the designer may introduce digital lookup tables (LUTs) into their design to implement user-defined functions on the analog hardware.

The device firmware provides a low-level programming interface that may be used to program an analog circuit to the analog device. The device firmware offers three kinds of settable digital values which together program the analog blocks:

- **Static Codes**: Static codes choose the function the analog unit implements. Static codes are combinatorial and do not represent decimal values. Setting static codes may affect the function implemented by the block and alter the block's physical characteristics. Static codes typically encode digitally settable bits that change the circuit structure. In this chapter, I will ascribe static code values to literals to improve readability.

- **Dynamic Codes**: Dynamic codes set constant digital values which appear in the function implemented by the block. Dynamic codes are digital integer values that map to decimal values. Dynamic codes typically are implemented as programmable current and voltage sources in the underlying analog circuit. Dynamic codes are also sometimes used to program digital logic elements such as LUTs. These logic elements may appear in analog circuits which offer programmable functions, for example.

- **Calibration Codes**: Calibration codes are internally set by the device runtime and firmware to eliminate any unwanted block behaviors. Calibration codes typically map to current and voltage sources in the underlying analog circuit. These current and voltage sources inject compensating signals into the analog circuit to remove unwanted behavior. The device firmware identifies the best set of calibration code values for the block at hand and the runtime writes the calibration code values to the device.

The device firmware also provides an interface for enabling and disabling digitally settable connections within the hardware.

This chapter presents an analog device specification language that offers a more natural abstraction for these low-level codes. I also present an analog device programming language that supports the configuration and connection of blocks at this higher level of abstraction. The compiler targets the analog device specification and produces an analog device program that implements the target dynamical system. The platform-specific runtime system then enables the analog device program connections in the firmware. It also translates the block configurations to static and dynamic code assignments and then writes these assignments to the device firmware. The runtime internally maintains the calibration code assignments.

The analog device specification language is made up of a collection of block specifications and a device layout specification. Each block specification defines block input ports and output ports, where each output port implements an input-output relation. Each block input and output may work with an analog current, an analog

voltage, or a continuously evolving digital signal. Each block specification encodes the static and dynamic codes with the following constructs:

- **Block Modes**: The block static codes are encoded as block modes. Each block mode corresponds to a set of static code assignments. The block specification language supports defining mode-dependent input-output relations.

- **Data Fields**: The dynamic codes belonging to a block are encoded as digitally settable data fields. The ADSL supports both constant data fields and data fields that implement expressions. Each data field is associated with an encoding scheme that specifies how to translate the decimal values written to each data field into integer values. The block data fields may appear in the input-output relations of the output ports.

The input-output relation implemented at each output port captures the *idealized* behavior of the block provided the block is not adversely affected by the low-level physics of the device.

5.2 Low-Level Physics and Analog Devices

Because analog devices directly leverage the underlying physics of the transistors, they are sensitive to the effects of process variations introduced during fabrication. These variations affect the low-level behavior of analog circuits on the device and alter the behavior of individual analog blocks on the hardware. The HCDCv2 is no exception and is also subject to these low-level physical effects:

- **Unexpected Signal Biases**: Signals within a circuit may experience unexpected offsets, called biases. If left uncorrected, these biases can significantly alter the function implemented by the analog block or introduce fixed biases into the output signals. Signals with bias issues produce errors that accumulate throughout the computation when integrated over time.

- **Unexpected Signal Gains**: Signals within a circuit may be scaled by random, non-unitary constant coefficients. These constant coefficients are referred to as signal gains. Unexpected signal gains within a block may drastically alter the function implemented by the analog block and introduce unexpected gains into the output signals. Signal gains also waste parts of the signal range.

- **Static Errors**: Some blocks experience static errors when exercised with certain inputs. These static errors may arise from nonlinearities in the circuit. These errors may manifest as point errors that occur in very small regions of the input space or adversely affect the output for large regions of the input space. These errors produce unexpected and difficult-to-model behaviors when the block is exercised with these inputs.

- **Unexpected Behaviors at High Frequencies**: Many analog blocks act as low-pass filters and attenuate away high-frequency signals. These errors may cause blocks to produce incorrect results for signals with fast-evolving frequency components.

- **Analog Noise**: All analog blocks are subject to analog noise. In the HCDCv2, the noise is typically randomly distributed around zero and can be adequately modeled with a Gaussian distribution. Analog noise introduces random perturbations into the computation – these perturbations may disproportionately affect the computation if the affected signal amplitudes are small.

To mitigate these issues, hardware designers have introduced a variety of mitigation strategies into the hardware design and documentation:

- **Device Calibration**: Hardware designers often strategically insert programmable current and voltage sources or introduce digitally configurable logic into their circuit designs. These programmable circuit components enable the designer to correct unwanted signal biases, unwanted signal gains, and static errors in the circuit after fabrication. Each circuit component can be configured by setting its respective `calibration code`. The device is calibrated before use to

eliminate as much unwanted behavior as possible. The *calibration* procedure performs a search over calibration codes to find the best set of codes. Because it is sometimes not feasible to eliminate all unwanted behavior, the hardware designer prioritizes eliminating the subset of behaviors that the compiler cannot automatically compensate for. For this reason, some hardware platforms offer multiple calibration strategies, each of which uses different criteria for selecting the best set of codes. The end user selects the calibration strategy before device calibration.

In practice, the analog blocks still may not behave as expected even after calibration. These post-calibration variations in behavior may change depending on the block's configuration and may vary across individual block instances. Examples of behavioral deviations which may persist post-calibration are unexpected biases and signal gains, and static errors.

- **Frequency Limitations**: Hardware designers identify the maximum supported frequency for each block during the initial characterization of the device. The identified frequency is the highest frequency signal a block can work with before frequency-dependent unexpected behaviors affect the computation. The compiler is then responsible for only providing signals within the supported frequency range.

- **Operating Range Limitations**: Hardware designers identify the minimum and maximum supported values at each block input and output during the design and initial characterization of the device. The minimum and maximum supported values together make up the operating range of the block input and output. Providing values outside of the port's operating range may damage the analog block or produce an incorrect result due to nonlinearities present in the underlying analog circuit. These operating range specifications are then provided to the end user.

  The compiler is then responsible for ensuring the programmed circuit doesn't violate any of these constraints.

- **Analog Noise**: Hardware designers identify the noise characteristics for each block during the initial characterization of the device. These noise measurements are then provided to the end user. The hardware designer may specify the noise characteristics as a symbolic distribution, a noise figure, or a simple standard deviation.

The compiler uses this information provided by the hardware designer to target the analog device effectively. Note that the compiler targets a specification of the analog device which encapsulates many of the above behaviors:

- **Delta Models and Compiler-Guided Compensation**: Some behavioral variations which remain post-calibration can be compensated for in compilation. For example, the compiler can correct for unexpected gains by selectively adjusting the data field values in the circuit. Other variations cannot be compensated for effectively. For example, the compiler can correct for unexpected biases by introducing digital-to-analog converters into the circuit that inject compensating analog currents into the affected ports. However, in practice, this correction technique introduces more error and drastically increases resource utilization. Other types of unexpected behaviors, such as static point errors, are difficult to model and cannot be compensated for effectively at compile-time.

  In this thesis, I introduce the concept of a delta model – a symbolic model which captures a subset of the unexpected behaviors present in the block after calibration. In this thesis, the delta models are used to capture unexpected gains and unexpected biases present in the calibrated blocks. The compiler then uses the block delta models to more effectively target the device on hand. The compiler compensates for unexpected gains by carefully scaling the circuit and adjusts for any unexpected biases introduced into data fields when assigning values to the block data fields. Section 5.3 provides a high-level overview of how delta models are elicited from the chip and used by the compiler.

- **Frequency Limitation Specifications**: The compiler-writer incorporates frequency limit annotations into the analog device specification. The compiler

adjusts the speed of the computation such that all of the frequency limit annotations are respected.

- **Operating Range Limitation Specifications**: The compiler-writer incorporates per-port operating range limit annotations in the analog device specification. The compiler adjusts the dynamic range of the signals in the circuit so that none of the specified operating range limitations are violated.

- **Noise Specifications**: The compiler-writer incorporates noise annotations in the analog device specification. In the analog device specification, the noise for each block port is defined as a standard deviation of the signal. The compiler uses the noise information to improve the signal-to-noise ratio at each of the ports. The compiler accomplishes this by selectively increasing the dynamic range of signals which are overcome by the noise floor.

Because the compiler handles all of the above phenomena, the end user is not required to reason about the low-level analog behaviors present in the device. The end user needs only to specify the calibration strategy the compiler should target. The end user can also optionally provide a minimum signal-to-noise ratio measure to limit to what degree the signals in the circuit are compressed.

5.3 Delta Models

Analog devices frequently do not perfectly implement the input-output relations described in the analog device specification post-calibration. These behavioral deviations may change depending on how the individual block is programmed. This thesis introduces the concept of a delta model, a mathematical model that captures the actual behavior of a calibrated, configured block. The block delta models are made up of two major components:

- **Delta Model Specification (Section 5.5.1)**: The analog device specification provides a delta model specification for each block. Each delta model specification is a templatized input-output relation over data fields, block ports, and

delta model parameters. The delta model parameters are variables that are resolved to constant values when targeting a specific block instance on the device. The delta model specification offers delta model parameter annotations which indicate which parameters can be compensated for in compilation and what each parameter value would ideally be if the block exhibited no behavioral variations.

Consider a multiplier block which ideally implements the function `c*x`. In this input-output relation, `c` is a data field and `x` is an input port. The delta model specification this multiplier is `(`α`*c+`β`)*x + `γ. In this specification, `c` is a data field, `x` is an input port, and α, β, and γ are delta model parameters. The ideal values for the α,β, and γ delta model parameters are `1`, `0`, and `0` respectively. With this instantiation, the multiplier implements `(1*c+0)*x+0` or `c*x` – this perfectly scales the signal by the constant value provided by data field `c`.

In practice, though, the delta model parameter values may deviate from their expected values. The compiler is able to effectively target blocks that have α and β parameters that deviate from `1` and `0`. The compiler can statically correct for the α and β parameters in compilation by carefully scaling the circuit and setting data field values. The delta model specification, therefore, annotates both of these parameters as correctable. The correctable annotation indicates that the compiler can statically handle variations in a delta model parameter value.

In contrast, the compiler cannot effectively correct the γ parameter. The compiler could potentially eliminate this γ parameter at the output port by summing the output signal with a signal implementing $-\gamma$. However, this is an expensive and ineffective correction in practice. To implement the $-\gamma$ signal, the compiler would need to introduce a digital-to-analog converter into the circuit – this block also introduces error into the computation. This compensation operation, therefore, increases the resource utilization of the circuit and introduces more error into the computation. For this reason, the γ parameter is deemed

uncorrectable.

- **Delta Model Database (Section 5.9.2)**: The device runtime populates the delta model parameters in the delta model specification for each block instance in the device on hand. The delta model parameter values for the target device are stored in the delta model database. The compiler uses the stored delta model parameters from the delta model database and the delta model specification from the analog device specification to target the device.

  The delta model database is populated offline by the runtime system. The runtime's profiling procedure exercises each block over a set of block inputs and stores the profiling data in the profiling database. The runtime's delta model elicitation procedure fits the profiling data to the delta model specifications defined in the device's ADS. The delta model parameters are the free variables in the model fitting procedure.

  For example, consider the case where the runtime has already identified the delta model parameters for the constant multiplier at location `0`. The runtime identified these parameters before compilation by profiling the block and fitting the delta model α`*c+`β`)*x + `γ to the collected data. The computed delta model parameters reside in the delta model database for the device on hand. or the multiplier at location `0`, the values of delta model parameters α, β, and γ are `0.95`, `0.012`, and `0.001` respectively.

  The compiler uses the α, β, and γ delta model parameters to concretize, or specialize, the delta model specification to describe the behavior of multiplier block `0` . The multiplier at location `0` implements `(0.95*c+0.012)*x+0.001` on the target device. The compiler then compensates for the α and β parameters at compile time. The compiler scales the input signal at port `x` of multiplier `0` by 1.053x to compensate for the α parameter of `0.95` when scaling the circuit. After scaling, the compiler shifts the value written to `c` by `-0.012` to account for the β delta model parameter of `0.012`.

Delta Models and Calibration

The delta model specification defines correctable delta model parameters which may deviate from their ideal values and uncorrectable delta model parameters which must closely adhere to their ideal values. The hardware designer can use this information to design calibration strategies that work well with the compiler. In this thesis, I explore two different types of calibration strategies:

- **Traditional Calibration**: The traditional calibration strategy calibrates the device such that all delta model parameters adhere to their idealized value. This calibration strategy produces calibrated blocks that implement the ideal input-output relations as accurately as possible. If the constant multiplier introduced above were calibrated with the traditional calibration strategy, the runtime could calibrate the block to implement `c*x`. This calibration strategy reflects how device calibration is typically performed for these kinds of hardware platforms.

- **Co-Designed Calibration**: The co-designed calibration strategy calibrates the device to allow for deviations in the delta model parameters that can be compensated for by the compiler. All other parameters must adhere to their idealized value. If the constant multiplier introduced above were calibrated with the co-designed calibration strategy, it would be configured to implement `(`α`*c+`β`)*x` where α and β may deviate from `1` and `0`. This calibration strategy prioritizes eliminating delta parameters and other behaviors that cannot be compensated for in compilation over delivering the ideal input-output relation.

The compiler uses the delta model specifications and the delta model database together to compensate for the behavioral variations present in the device on hand.

5.4 Notation for Language Grammars

Figure 5.1 presents the shorthand notation used by the language grammars described in this chapter. These notational shortcuts are used to reduce the complexity of the

shortcut	expanded rule	
body(<r>)	body<r>	::= <r> | body<r>;<r>
seq(<r>)	seq<r>	::= <r> | seq<r>,<r>
tup(<r>)	tup<r>	::= (seq(<r>))
lst(<r>)	lst<r>	::= [seq(<r>)]
pat(<r>)	pat<r>	::= | <r> | pat(<r>) | <r>
match(<r1>,<r2>)	match<r1><r2>	::= (pat(<r1>)-><r2>)*
multi(<r1>,<r2>)	multi<r1><r2>	::= <r2> | func match(<r1>,<r2>)

Table 5.1: Shorthand for common syntactic entities

language grammars. Each of these shorthand functions accepts one or more symbols and produces a set of rules which implement the desired syntactic entity. These constructs generate rules for sequencing (`seq`), tuples and lists (`tup`, `lst`), and pattern matching (`pat`,`match`, `multi`). I summarize each of these convenience functions below:

- **Semicolon-Delimited Statements**: The `body(<r>)` function implements a semicolon-delimited sequence of the symbol `<r>`. The `body(<r>)` function injects the `body<r> = <r> | body<r>; <r>` rules in the language grammar. After expansion, the `body<r>` symbol is inserted everywhere where the convenience function was initially used.

- **Comma-Delimited Statements**: The `seq(<r>)` convenience function implements a comma-delimited sequence of the symbol `<r>`. For example, the `seq(E)` function inserts the `seqE ::= E | seqE; E` rule into the language grammar. After expansion, the `seq<r>` symbol is inserted everywhere where the function was initially used.

- **Tuples**: The `tup(<r>)` function implements tuples made up of the symbol `<r>`. This function makes use of the `seq(<r>)` shortcut to implement the comma-delimited list of `<r>` elements. The `tup(<r>)` shortcut inserts the `tup<r> ::= [ seq<r> ]` and the `seq<r> ::= <r> | seq<r> , <r>` rules into the language. After expansion, the `tup<r>` symbol is inserted everywhere where the function was initially used.

  For example, the `Q ::= tup(E)` invocation adds the `tupE ::= ( seqE )` rule, the `seqE ::= <r> | seqE , E` rule, and the `Q ::= tupE` rule. The `Q` symbol recognizes strings such as `(a+b,x*y)` and `(w*w)` after the expansion.

- **Lists**: The `lst(<r>` function implements comma-delimited lists made up of the symbol `<r>`. This function makes use of the `seq(<r>)` convenience function to implement the comma-delimited list. The `lst(<r>)` function inserts the `lst<r> ::= [ seq<r>`

] and the seq<r> ::= <r> | seq<r> , <r> rules into the language grammar. After expansion, the lst<r> symbol is inserted everywhere where the function was initially used.

For example, the Q ::= lst(E) invocation adds the lstE ::= [seqE] rule, the seqE ::= <r> | seqE , E rule, and the Q ::= lstE rule. The Q symbol recognizes snippets such as [a+b,x*y] and [w*w] after the expansion.

- **Patterns**: The pat(<r>) function implements pipe-delimited sequences of the symbol <r>. The pat(<r>) function inserts the pat<r> ::= | <r> | pat<r> | <r> rule into the language grammar. Each invocation of pat(<r>) is replaced with the symbol pat<r> after the function is expanded.

  For example, the Q ::= pat(E) invocation adds the patE ::= | E | patE | E rule and the Q::=patE rule to the grammar. The Q symbol recognizes snippets such as |a+b | x*y and | q.

- **Pattern Matching** The match(<r1>,<r2>) function implements sequences of pattern matching statements of the form <r1> <r2> for the symbols <r1> and <r2>. This symbol expands the pat(<r1>) convenience function to build the sequence of pipe-delimited <r1> symbols and adds the match<r1><r2> ::= (pat(<r1>)-><r2>)* rule into the language grammar. Each match(<r1>,<r2>) invocation is replaced with the match<r1><r2> symbol after the shortcuts are expanded.

  For example, the Q::= match(E,I) rule expands to the following set of rules after the match function is expanded to the patE ::= | E | patE | E rule, the matchE ::= (patE -> I)* rule, and the Q::= matchEI rule. The Q symbol is able to recognize snippets such as | x + v -> [0,1] | v -> [0,2].

- **Parametric Statements**: The multi(<r1>,<r2>) shortcut implements a symbol which is either the symbol <r2> or a pattern matching statement over the <r1> symbol which evaluates to the <r2> symbol. This function expands the pat(<r1>) and match(<r1>,<r2>) functions to build the match statements and adds the match<r1><r2> ::= <r2> | func match<r1><r2> rule into the language grammar. Each shortcut invocation multi(<r1>,<r2>) is replaced with the multi<r1><r2> symbol.

5.5 Analog Device Specification Language

The compiler works with a specification of the target analog device that captures the functional behavior, programming interface, and the physical limitations and behaviors present within the device. This language can be used to specify a variety of reconfigurable, differential equation-solving analog devices. In this thesis, I focus on the analog device specification for the HCDCv2 analog device. This analog device specification (ADS), written in the analog device specification language (ADSL), comprises a collection of block specifications and an analog device layout specification:

```
Spec ::= (BlockSpec)*DeviceSpec
```

Each analog device specification (ADS) provides a formal block specification (`BlockSpec`, Section 5.5.1) for each kind of block resident on the target device. The analog device specification also provides a device layout specification (`DeviceSpec`, Section 5.5.2) which defines all the individual instances of each kind of block and how the defined block instances may be connected together. The device layout specification also defines how these block instances are spatially laid out on the analog hardware.

The compiler produces as output an analog device program (ADP) written in the analog device programming language (ADPL). The ADP describes a mixed-signal circuit comprised of configured blocks. The analog device program configures one or more block instances and routes these blocks together with digitally programmable connections. The produced ADP is executable and can be run on the analog device described by the input ADS.

The generated ADP specifically targets the ADS provided to the compiler. The ADP will only configure blocks with an ADS block specification and only make connections that are listed in the device specification. Each ADP block configuration only writes values to variables that are part of the programming interface of the block.

This section will formally describe the analog device specification language and the analog device programming language. These specification languages will be used in Section 5.7 to formally describe the programming interface and available blocks for the HCDCv2 device. Both languages use the mathematical expressions introduced in Section 3.2.1 and the grammar convenience functions introduced in Section 5.4.

```
Mode = tuple(l), ModeR = tuple(l|*)
BlockT  ::= compute | assemble | route
DeltaT  ::= gain | offset | other
AnalogT ::= current | voltage
SigT    ::= analog AnalogT | digital
DataT   ::= const | expr lst(v)
PortT   ::= in | out
QuantT  ::= linear d
IFace   ::= PortT seq(v) sigT (extern)? | data v DataT
DeltaM  ::= delta-par seq(v) (correctable)? DeltaT ideally x
         | delta lst(v) =  multi(ModeR,E)
Impl    ::= rel v = multi(ModeR,E)
         | interval seq(v) = multi(ModeR,I)
         | quantize lst(v) = multi(ModeR,QuantT)
         | maxfreq lst(v) = multi(ModeR,n) | noise lst(v)  multi(ModeR,x)
         | DeltaM
Def ::= block v BlockT modes lst(Mode)
Stmt ::= IFace | Impl
BlockSpec ::= Def {body(Stmt)}
```

Figure 5-1: ADSL block specification grammar.

5.5.1 Block Specification Language

The block specification language captures the behavior of each block on the analog device. The block specification language uses the basic and extended expressions presented in Section 3.2.1 to formally describe block behavior. Figure 5-1 presents the block specification language. Each block specification is made up of a block definition (`Def`) and a collection of block interface (`Iface`) and implementation (`Impl`) statements.

Block Definition(`Def`): Each block definition `block v BlockT modes lst(Mode)` specifies the block name `v` and type `BlockT` then lists all of the possible modes lst(Mode) which may be configured to the block.

- *Block Type* (`BlockT`): The block type indicates what the specified block is used for. Each block either computes (`compute`), copies and converts signals (`assemble`), or routes signals (`route`) through the chip.

  The block type influences the kinds of computations which are allowed in the block specification. The `compute` blocks are not subject to any restrictions on the implemented computation. In comparison with `compute` blocks, `assembly` and `route` blocks are specialized blocks with additional constraints. `assemble` blocks may only copy or

negate signals with only unity (`-1` or `1`) coefficients. The `route` blocks have a single mode, only one input port and one output port, and cannot perform any computation on the input signals. `route` blocks exist to enable successful signal routing in the presence of constraints on connections between output and input ports from different blocks.

- *Block Modes* (`lst(Mode)`): Each block is parametric and can be placed in one of many possible block modes. The block modes set the behavior and the physical restrictions imposed on the block. Each block mode is described as a tuple of literals.

Block Interface(`Iface`): Each block has a set of associated input and output ports. Block ports may be routed together to form a circuit. Each block port works with one kind of time-varying signal. The type of signal may impose limitations on how the block ports may be routed together. Analog currents are added together by joining wires and cannot be used more than once without the aid of a copier block. Analog voltages must be added together with a dedicated block but can be used more than once without any special hardware.

Blocks may also be programmed by digitally setting block data fields. The block mode and block data fields together make up the programming interface of the block. The block interface statements together specify all of the block ports and data fields:

- **Input Block Ports**: Each block may have any number of input block ports. Each input port accepts a time-varying signal of a specific signal type. Each input port definition statement `in seq(v) sigT (extern)?` declares one or more named input ports seq(v) with the `synin` port type and the specified signal type `sigT`. Supported signals include analog currents (analog current), analog voltages (analog voltage), and time-varying digital signals (digital). Some input ports may be externally accessible (`extern`) External input ports accept externally provided signals from other devices, such as sensors.

- **Output Block Ports**: Each block has one or more output ports. Each output port produces a time-varying signal of a specific signal type. This output port carries the result of whatever computation the block is designed to implement. Each output port definition statement `out seq(v) sigT (extern)?` declares one or more named ports seq(v) with the `synout` port type the specified signal type `sigT`. The output

ports support the same signal types as the input ports. Some output ports may be may be externally accessible (`extern`). External output ports are linked to externally accessible pins on the device may be observed with an external measurement device such as an oscilloscope.

- **Constant Data Fields**: Some blocks offer digitally programmable constant data fields which resolve to constant decimal values during execution. These constant data fields influence the input-output relation implemented at each block output port. Each constant data field definition `data v const` declares a named data field `v` with the `const` data type.

- **Expression Data Fields**: Some blocks offer digitally programmable expression data fields which can be used to provide user-defined functions to a block. Expression data fields typically define part of the input-output relation implemented at each block output port. Each expression data field definition `data v expr lst(v)` declares a named data field `v` with the `expr` data type which accepts the variable arguments lst(v). Each defined expression data field only accepts expressions which use variables in the provided argument list.

Block Implementation(`Impl`): Each block output implements an input-output relation over block inputs and data fields. Ideally, the signal at each block output evolves in accordance with the block output's input-output relation. I refer to this as the *ideal* behavior of the block. In practice, each block is subject to noise, quantization error, and process variation and must work with signals which do not violate the block's operating range and frequency restrictions. The block implementation statements specify the input-output relation for each output and describe all of the physical limitations and behaviors which must be considered during compilation:

- **Input-Output Relation**: Each block output implements a parametric input-output relation which changes depending on the block mode. This input-output relation may involve any of the block data fields and input ports defined in the block interface. Each provided expression may involve the input ports and data fields specified in the block interface.

  The block `rel` statements capture the behavior of the output port in the absence

of manufacturing variations, noise, and quantization error. Each `rel` statement defines the expression implemented at each output port under each mode. The relation statement `rel v = multi(ModeR,E)` specifies the mode-dependent expression `multi(ModeR,E)` implemented at the output port `v`.

- **Operating Range Restrictions**: Each port and constant data field may only take on a limited range of values. Analog ports may only accept values within their respective current and voltage operating ranges. Supplying values outside these ranges may damage the block or cause the block to behave incorrectly. Digital ports and data fields can only encode values that fall within the supported range of values. The range of supported values for a particular port or data field may change depending on the block mode.

  All operating range restrictions are defined in the block specification with `interval` statements. Each `interval` statement describes the range of values supported at a port or data field. The operating range restriction `interval seq(v) = multi(ModeR,I)` specifies a mode-dependent interval `multi(ModeR,I)` which captures the range of values supported at the port or data field `v` under each block mode. Note that `interval` statements can also be written for expression data field arguments.

- **Frequency Restrictions**: Blocks may also require the device to run computations at a lower speed to operate properly. Analog blocks often act as low-pass filters and attenuate away higher frequency components of a signal. Blocks that work with analog and digital signals internally sample analog signals at a specific rate and cannot process signals frequency components that exceed the Nyquist frequency.

  These frequency restrictions are expressed as upper limits on the maximum frequency of the device (`maxfreq`). Each maximum frequency restriction statement `maxfreq lst(v) = multi(ModeR,n)` specifies a mode-dependent maximum supported frequency `multi(ModeR,n)` for the listed set of block ports `lst(v)`. These maximum frequency restrictions apply if the any of the listed ports are in use (connected to another port).

- **Noise**: Blocks with analog signals often introduce analog noise into the computation. This noise may be further amplified depending on the mode of the block. The `noise` statements describe the standard deviation of the signals at the analog

ports. Each noise statement specifies `noise lst(v) = multi(ModeR,x)` the mode-dependent standard deviation `multi(ModeR,x)` of the signal at the analog ports `lst(v)`.

- **Quantization Error**: Blocks with digital ports and data fields are subject to resolution limitations and quantization error. The `quantize` statements describe the value encoding scheme used at each digital port and data field. The `quantize` and `interval` statements together determine the quantization error of a digital port or data field. Each quantize statement `quantize lst(v) = multi(ModeR,QuantT)` statement specifies the mode-dependent quantization strategy `multi(ModeR,QuantT)` for the list of digital ports and data fields lst(v). Currently, only linearly encoded digital values are supported. The `linear d` clause indicates the digital signal is divided into `d` equally spaced segments.

- **Delta Model**: In practice, fabricated instances of the block may implement variations of the described input-output relation. These variations in behavior are typically eliminated through a process called calibration. However, it is not always possible to eliminate all of these variations in behavior. For this reason, all remaining variations in behavior are exposed at compile-time through the delta model specification.

  The delta model specification of the block (`delta-par` and `delta` statements) describes all the functions which may be implemented at an output port in practice after the block has been calibrated. Each delta model contains one or more delta model parameters. Delta model parameters may be correctable or uncorrectable ((correctable)?). Correctable delta model parameters can be statically compensated for at compile-time, while uncorrectable delta model parameters cannot. Each delta model parameter also has a parameter type (`gain`, `offset`, or `other`) which indicates what kind of unwanted behavior is captured with the parameter. The compiler uses the parameter type to further narrow down the correctable delta model parameters that should be compensated for by a given compilation pass. All delta model parameters have an ideal parameter value `ideally x`. Instantiating the delta model parameter to its ideal parameter value typically eliminates the effect of the parameter on the block dynamics.

  Correctable Delta Model Parameters: Correctable delta model parameters can be com-

pensated for at compile-time. The correctable delta parameter definition statement `delta-par seq(v) correctable DeltaT ideally x` defines a list of correctable delta model parameters `seq(v)` which have the type `DeltaT` and ideally would take on the value `x`.

Uncorrectable Delta Model Parameters: Uncorrectable delta model parameters cannot be corrected at compile-time. The delta parameter definition statement `delta-par seq(v) DeltaT ideally x` defines a list of uncorrectable delta model parameters `seq(v)` which have type `DeltaT` and should ideally should be close to the real value `x`.

The Delta Model Relation: Each analog output port is optionally assigned a delta model specification. The delta model specification is a mode-dependent input-output relation over data fields, block inputs, and delta model parameters. Each `delta lst(v) = multi(ModeR,E)` statement defines a mode-dependent mathematical expression `multi(ModeR,E)` which describes the space of delta models at the output ports `lst(v)`.

Block `interval` statements are required for all ports and data, `quantize` statements are required only for digital ports, and `maxfreq` and noise statements are optional. Together, the `interval` and `maxfreq` specify the *physical restrictions* of the block. These statements impose hard constraints on the block which must be honored for hardware to operate correctly. The `noise`, `quantize`, and `delta` statements specify the *physical behaviors* of the block. These statements specify unwanted behaviors which may alter the accuracy of the signal produced at each block input. Typically, these behaviors cannot be completely corrected, but can be attenuated away through careful programming.

5.5.2 Device Layout Specification

Figure 5-2 presents the device layout specification language, which identifies the blocks and connections available on the analog device. The device specification defines a collection of locations `Loc` on the analog device. Each device location may contain at most one of each kind of block. Each block instance is therefore uniquely identified by the block location and the block name. Each device location `Loc` is encoded as a tuple of numbers (`tuple(n)`).

```
Loc = tuple(n)
PatLoc = tuple(*|n)
Ports = seq(l) @ seq(PatLoc) (port v)?
Stmt ::= struct lst(n) in v
       | views seq(v) | freq n
       | blk seq(l) @ PatLoc
       | conn Ports1 with Ports2

DeviceSpec ::= device v {block(Stmt)}
```

Figure 5-2: ADSL device layout specification grammar

Analog hardware designers provide more digitally programmable connections for spatially co-located locations than for spatially distant locations. It is therefore important for the device specification to also describe the spatial orientation of each location on the analog device. The device specification organizes the spatial structures of the device into sequentially organized views. The first view is the most general view which contains the largest structures on the device. The subsequent views capture successively finer grain device structures. Each spatial structure in a view $\mathtt{v}_i$ is uniquely identified with a spatial location tuple of $i+1$ numbers. Block instances may only be bound to spatial locations belonging to the finest grain view – this corresponds to the last view in the specification. The spatial locations from the finest grain view are simply referred to as locations.

Each view v_i partitions the spatial locations from the parent view v_{i-1} into one or more substructures. The location tuple $(\mathtt{n}_0,..,\mathtt{n}_i,..,\mathtt{n}_m)$ captures each structure the location belongs to in each view. Given a view $\mathtt{v}_i$, the location tuple belongs to the structure at the spatial location (n_0,..,n_i). The structure at spatial location $(\mathtt{n}_0,..,\mathtt{n}_{i-1},\mathtt{n}_i)$ is inside the parent structure (n_0,..,n_i-1) from the previous view $\mathtt{v}_{i-1}$. The integer identifier $\mathtt{n}_i$ identifies the the structure within the parent structure $(\mathtt{n}_0,..,\mathtt{n}_{i-1})$. The distance between two spatial locations can therefore be approximated by computing the length of the shared prefix between the location tuples.

The following language constructs specify the spatial layout of the device locations and map blocks to locations:

- **Views**: The view specification statement `views lst(v)` provides an ordered list of named views `lst(v)` for the analog device. The most general (first) view contains structures with spatial locations of length 1 and the most specific (nth) view contains

spatial locations of length n.

- **View Structures**: Each view is made up of a collection of non-overlapping structures. The `struct lst(n) in v` statement defines the set of integers txlst(n) which identify strctures for a view `v`. The i^{th} value of any spatial location must belong to `lst(n)` to be valid.

- **Block Instances**: The `blk` statements declare new block instances. Each block instance declaration `blk seq(l) @ seq(PatLoc)` statement maps blocks `seq(l)` to device locations which match the location pattern seq(PatLoc). The location pattern `PatLoc` is a tuple of integers `n` and wildcard symbols `*`. The location pattern only accepts locations which match all of the integer identifiers in the tuple. For example, the location pattern `(3,*,2,*)` would match location `(3,3,2,0)` but not location `(3,3,1,0)`.

- **Connections**: The `conn` statements declare sets of connections between the ports of block instances. Each connection statement `conn` $\texttt{Ports}_1$ `with` $\texttt{Ports}_2$ adds a set of digitally programmable connections which link all of the output ports in $\texttt{Ports}_1$ with all the input ports in $\texttt{Ports}_2$. The statement adds connections for pairs of input and output ports which have the same signal type.

  Each port collection `Port` specifies a set of input or output port instances. A *port instance* is uniquely identified by the block name and location and the port name. Each port collection clause `seq(l) @ seq(Loc) (port v)?` contains all of the ports belonging to block instances whose block name is in `seq(l)` and block location is in `seq(Loc)`. If the `(port v)?` clause is specified, the collection only contains port instances with the port name `v`.

- **Hardware Time Constant**: The frequency statement `synfreq x` defines the baseline integration speed of the device `x` in hertz. The hardware time constant is the reciprocal of the baseline integration speed ($\texttt{x}^{-1}$). The hardware time constant specifies the amount of wall-clock time which corresponds to one unit of integration time.

```
Loc = tuple(n)
Mode = tuple(l)
Port = block l1 port l2 @ Loc
Cfg ::= set l = x | set l = F | modes lst(Mode) |scale l = x | source E at l
AStmt ::= config block l @ Loc {body(Cfg) }
          |conn Port1 with Port2
          |timescale x
ADP ::= body(AStmt)
```

Figure 5-3: Grammar for analog device program language (ADPL)

5.6 Analog Device Programming Language

An analog device program (ADP) specifies a configuration of the analog device as generated by the compiler from an input ADS and DSS. At a high level, the ADP configures the analog device specified in the ADS to implement a mixed-signal circuit comprised of configured blocks. This mixed-signal circuit implements the provided DSS. The ADP is loaded into the device via a platform-specific runtime system.

The ADP optionally specifies the scaling transform, which scales the computation to ensure it respects the physical constraints of the device. The scaling transform is composed of a collection of magnitude scaling factor assignments for the ports and data fields in the ADP and time scaling factor, which describes the speed of the computation. An ADP which defines a scaling transform is called a *scaled* ADP. An ADP which does not define a scaling transform is called a *unscaled* ADP. Figure 5-3 presents the analog device programming language:

- **Connections**: The analog device program forms a circuit by specifying which digitally settable connections to enable on the analog device. Each `conn` $\texttt{Port}_1$ `with` $\texttt{Port}_2$ statement connects the input port $\texttt{Port}_1$ with the output port $\texttt{Port}_2$. Each port declaration `block` $\texttt{l}_1$ `port` $\texttt{l}_2$ `@ Loc` references the port $\texttt{l}_2$ from the block instance $\texttt{l}_1$ at location `Loc`. Each connection statement only connects together ports which have a defined connection in the ADS.

- **Block Configurations**: The block configuration (`config`) statements digitally configure block instances so that they implement the desired expressions. It specifies the name and location of the block instance, values for digitally settable fields, and the set of viable modes for that block. The block configuration statement `config block l @`

`Loc {body(Cfg) }` writes the block configuration `body(Cfg)` to the block instance with the name `l` at location `Loc`. The block configuration `Cfg` instantiates the data fields and modes defined in the ADS block specification for block `l`. The data fields and modes together make up the programming interface of the block:

Constant Data Field Assignments: Each constant data field assignment `set l = x` sets the constant data field `l` to the real value `x`.

Expression Data Field Assignments: Each expression data field assignment `set l = F` instantiates an expression data field `l` to the extended expression `F`. The expression `F` may only reference the listed variable arguments for expression data field `l` as specified in the ADS.

Mode Assignments: The mode assignment statement `modes lst(Mode)` selects the subset of modes `lst(Mode)` for the configured block instance. A block configuration is considered *complete* if only one viable mode is listed for each block.

Source Annotations: Source annotations map physical signals to dynamical system variables and expressions. Each source annotation `source E at l` specifies that the signal at port or data field `l` implements the dynamical system expression `E`. The dynamical system expression `E` may only contain DSS variables.

Magnitude Scale Factors: Each magnitude scale factor declaration `scale l = x` assigns the magnitude scale factor value `x` to the port or data field `l`. During execution, the runtime multiplies each block data field value by the associated magnitude scale factor. This applies the scaling transform to the circuit.

- **The Time Scale Factor**: The `timescale x` statement specifies the time scaling factor `x` of the scaled ADP. The runtime multiplies the time scale factor with the ADS time constant to compute the mapping between wall-clock time and simulation time for the scaled computation. The runtime uses this mapping to compute the runtime of the computation.

5.7 HCDCv2 Analog Device Specification

This section presents the analog device specification of the HCDCv2. Figure 5-4 presents a picture of the HCDCv2 analog device. The HCDCv2 analog device provides integration

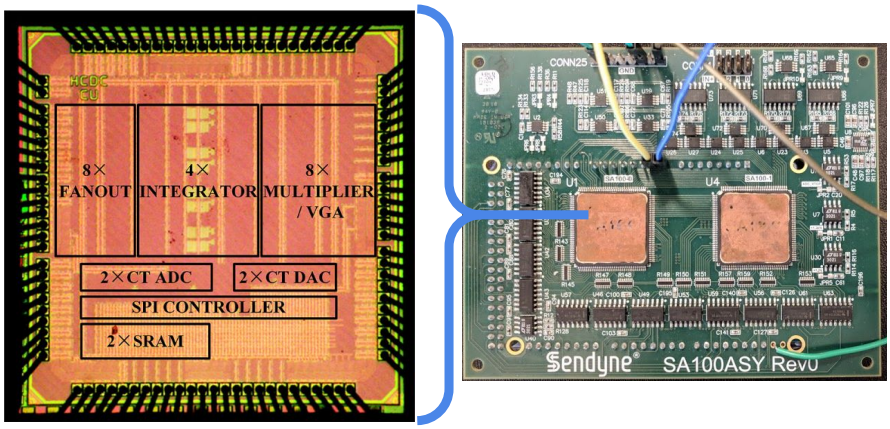


Figure 5-4: Die photo of HCDCv2 chip and HCDCv2 Analog Device [51]

and multiplication analog blocks, current copier blocks, and digital-to-analog converters and analog-to-digital converters. The HCDCv2 is a current-mode programmable analog device with a baseline integration speed of 126000 Hz. At this speed, one unit of integration time corresponds to 7.93 μs of wall clock time. The hardware time constant is therefore `7.93` μs. The HCDCv2 represents continuously evolving program values using three types of signals:

- **Analog Currents**: The majority of the signals in the HCDCv2 are analog currents. The HCDCv2 typically works with currents between [-2,2] μA but can be configured to work with currents as large as [-20,20] μA.
- **Analog Voltages**: All externally accessible signals on the HCDCv2 are analog voltages ranging from [-1.2,1.2] V. These voltages are easier to measure reliably than the analog currents, which are internally used to perform the computation.
- **Digital Signals**: Some blocks work with continuously evolving digital signals. These digital signals are eight bit values (between 0-255) which map to decimal values between [-1,0.998]. Given a decimal value `x`, the digital encoding of this value is `x*128 + 128`. In this section, the `todec` function maps HCDCv2 digital values to decimal values and the `fromdec` function maps decimal values to HCDCv2 digital values.

Compilation and Execution:The compiler targets the HCDCv2 ADS and produces ADPs which can be written to the HCDCv2. The HCDCv2 runtime then executes the ADPs generated by the compiler and records all externally accessible signals to files. The HCDCv2 runtime also provides functions for calibrating, profiling, and eliciting delta models

from the device. The runtime maintains a calibration database that stores calibration information, a profiling database that stores profiling information, and a delta model database that stores the delta model parameters for each block on the device. All of these procedures are performed offline before compilation.

5.7.1 HCDCv2 Block Specifications

The HCDCv2 has 6 types of computational blocks (`mul`, `adc`, `dac`, `int`, `extin`, `extout`, and `lut`), 4 types of route blocks (`tout`, `tin`, `cout`, `cin`), and 1 type of copier block (`fan`).

The low-level programming interface of the HCDCv2 blocks is made up of static, dynamic, and calibration codes. Each HCDCv2 block is programmed by writing a collection of positive, integer-valued, digital codes to the block's memory. The codes are summarized below:

- **Static Codes**: The block's static codes change the block input-output relations and influence the operating range, noise characteristics, and frequency limitations of the block. The ADS encodes each combination of static code values are encoded as a distinct block mode. The compiler selects the block modes during compilation – this instantiates the static codes for each block.

- **Dynamic Codes**: The block dynamic codes map to programmable decimal values into the block input-output relations. Each dynamic code is an 8-bit value that maps to decimal values between -1 and 0.998. The dynamic codes use the same encoding scheme as the digital signals presented at the beginning of Section 5.7. The ADS encodes each block's dynamic codes as constant and expression data fields in the ADS. The compiler instantiates constant and expression during compilation – this instantiates the dynamic codes for each block.

- **Calibration Codes**: The block calibration codes are used to calibrate the block. The HCDCv2 firmware deploys calibration routines that identify the best set of calibration code values for each block. Refer to Section 5.8 for a discussion of the HCDCv2 calibration routines. The HCDCv2 runtime calibrates the HCDCv2 blocks offline (before compilation) and stores the calibration code values in the calibration code database. The HCDCv2 writes the stored calibration code values to the HCDCv2

```
block int type compute modes [(+,m,m),
    (+,m,h),(+,h,m),(+,h,h),(-,m,m),
   ,(-,m,h),(-,h,m),(-,h,h)] {
   in x analog current; out z analog current;
   data z0 const;
   rel z = func |(h,m,+) -> integ(0.1*x,2*z0)
        |(m,m,+) -> integ(x,2*z0)  |(h,h,+) -> integ(x,20*z0)
        |(m,h,+) -> integ(10*x,20*z0)
        |(h,m,-) -> -integ(0.1*x,2*z0)
        |(m,m,-) -> -integ(x,2*z0)  |(h,h,-) -> -integ(x,20*z0)
        |(m,h,-) -> -integ(10*x,20*z0)
   delta-par a,b correctable gain ideally 1;
   delta-par c correctable offset ideally 0;
   delta-par u offset ideally 0;
   delta z = func |(h,m,+) -> integ(0.1*a*x+u,2*(b*z0+c))
        |(m,m,+) -> integ(a*x+u,2*(b*z0+c))
        |(h,h,+) -> integ(a*x+u,20*(b*z0+c))
        |(m,h,+) -> integ(10*a*x+u,20*(b*z0+c))
        |(h,m,-) -> -integ(0.1*a*x+u,2*(b*z0+c))
        |(m,m,-) -> -integ(a*x+u,2*(b*z0+c))
        |(h,h,-) -> -integ(a*x+u,20*(b*z0+c))
        |(m,h,-) -> -integ(10*a*x+u,20*(b*z0+c))
   quantize z0 = linear 256;
   max-freq z = 80000;
   noise x = func | (m,*,*) -> 0.02 | (h,*,*) -> 0.2
   noise z = func | (*,m,*) -> 0.02 | (*,h,*) -> 0.2
   interval z0 = [-1,0.998];
   interval z = func |(*,m,*) -> [-2,2] |(*,h,*) -> [-20,20];
   interval x = func |(m,*,*) -> [-2,2] |(h,*,*) -> [-20,20];}
```

Figure 5-5: Block specification for integrator

before executing each ADP.

This section presents the block specifications for the HCDCv2 analog device and summarizes the static, dynamic, and calibration codes for each block. All of the block specifications presented in this section define delta models – the compiler uses these delta model specifications and the delta model parameters defined in the delta model database to compensate for behavioral deviations during compilation. Refer to Section 5.9 for more information on how the HCDCv2 software stack elicits delta models.

Integrator (`int`) Block

The HCDCv2 provides an integrator (`int`) block which integrates an analog signal over time. The `int` block accepts an analog current input at port `x` and produces an analog current output at port `z`. It has one dynamic code (`z0`) and three static codes (`in`,`out`,`sgn`).

The static codes `in` and `out` may either be assigned m or h. The code `sgn` is either `+` or `-`. Figure 5-5 presents the ADP block specification for the integrator (`int`) block. The `int` block is defined as a `compute` block with 11 modes.

Modes: Each combination of static code values is encoded as a distinct ADP block mode. Each block mode used in the specification is `(in,out,sgn)`. The integrator block works with one analog input `x`, one constant data field `z0`, and one output `z`. The constant data field `z0` is mapped by the HCDCv2 runtime to the dynamic code `z0`.

Functionality:The `rel` statement describes input-output relation implemented at the output port `z` of the integrator block under each mode. The block generally integrates the current at input port `x` starting with an initial value determined by dynamic code `z0`. Depending on the mode, the block may introduce constant coefficients into the implemented expression or negate the output.

The block specification also includes a delta model specification (lines 12-18) which captures the space of behavioral deviations in the fabricated integrator blocks. The multiplier block has three correctable delta model parameters (`a`, `b`, `c`) and one uncorrectable delta model parameter (`u`). The `a` parameter scales the derivative and the `b` and `c` parameters linearly transform the initial condition. The `u` parameter is the offset of the derivative and cannot be easily compensated for in compilation. The delta parameter `u` ideally would be close to zero. The delta model parameters `a`, `b`, and `c` parameters can be compensated for by the compiler. The `a` and `b` parameters are compensated for by the `LScale` pass of compilation and the `c` parameter is compensated for by the HCDCv2 runtime.

The block mode influences the current range restrictions imposed on ports `x` and `z`. For example, when `in` is m, the port accepts currents between [-2,2] μA. When `in` is h, the port accepts currents between [-20,20] μA. The `z0` data field uses the same value encoding scheme as the digital signals presented at the beginning of Section 5.7. The block mode also affects the noise characteristics of the block. The analog currents at ports `x` and `z` are subject to more noise when static codes `in` and `out` are set to h.

The integrator block limits the simulation speed to 80 kHz. This frequency limitation is necessary for ensuring the integrator isn't forced to operate in a regime where the frequency-gain characteristics of the signal become complex.

Calibration Codes: The `int` block also works with 5 calibration codes (`biasIn`, `biasOut`, `pmos`, `nmos`, `gainCal`). The `biasIn` and `biasOut` codes take on values between 0 and 63.

```
block mult type compute modes [(m,m,m),
    (m,m,h),(h,m,h),(m,h,h),(h,h,h),
   ,(x,m,m),(x,m,h),(x,h,m),(x,h,h)] {
   in x,y analog current; out z analog current;
   data c const;
   rel z = func |(m,m,h) -> 5*x*y,
        |(m,m,m)|(h,m,h)|(m,h,h) -> 0.5*x*y
        |(h,m,m)|(m,h,m)|(h,h,h) -> 0.05*x*y
        |(x,m,h)->10*c*x |(x,h,m) -> 0.1*c*x
        |(x,h,h)|(x,m,m) -> c*x;

   delta-par u correctable gain ideally 1;
   delta-par v correctable offset ideally 0;
   delta-par w offset ideally 0;
   delta z = func |(m,m,h) -> u*5*x*y+w,
        |(m,m,m)|(h,m,h)|(m,h,h) -> u*0.5*x*y+w
        |(h,m,m)|(m,h,m)|(h,h,h) -> u*0.05*x*y+w
        |(x,m,h)->10*c*x |(x,h,m) -> 0.1*(u*c+v)*x+w
        |(x,h,h)|(x,m,m) -> (u*c+v)*x+w;

   quantize c = linear 256;
   max-freq z = func |(m,*,*) -> 40000 | (h,*,*) -> 40000 | (x,*,*) -> 126000
   noise z = func | (*,*,m) -> 0.02 | (*,*,h) -> 0.2
   noise x = func | (*,m,*) -> 0.02 | (*,h,*) -> 0.2
   noise y = func | (m,*,*) -> 0.02 | (h,*,*) -> 0.2
   interval c = [-1,0.998];
   interval z = func |(*,*,m) -> [-2,2] |(*,*,h) -> [-20,20];
   interval x = func |(*,m,*) -> [-2,2] |(*,h,*) -> [-20,20];
   interval y = func |(h,*,*) -> [-20,20] |(*,*,*) -> [-2,2]; }
```

Figure 5-6: Block specification for multiplier

The `pmos` and `nmos` codes take on values between 0 and 7. These calibration codes are used to eliminate bias in the block and tune the block to achieve unity gain.

Multiplier (`mul`) Block

The HCDCv2 provides a multiplier (`mul`) block which scales signals and multiplies signals together. The `mul` block accepts two analog current inputs (ports `x` and `y`) and produces one analog current output (port `z`). It has one dynamic code (`c`) and four static codes (`in0`, `in1`, `out`, and `vga`). The static codes `in0`, `in1`, and `out` may either be assigned m or h. The `vga` static code is either true or false.

Figure 5-6 presents the block specification for an analog multiplier. The `type` clause identifies the multiplier as a `compute` block. The block has eleven modes, two analog current inputs (`x` and `y`), one constant data field (`c`), and one analog current output (`z`). The constant

data field `c` corresponds to the the dynamic code `c` in the `mul` analog block. The block mode and constant data field makes up the programming interface for the block.

Modes: The `modes` define modes `(m,m,m)` through `(x,h,h)`. The block modes encode all the possible combinations of static code assignments for the multiplier block:

if `vga` = `true` **then** `(x,in0,out)` **else** `(in1,in0,out)`

The above mode representation encodes each combination of static code values as a tuple of literals. The value of the `in1` static code is included in the encoding if `vga` is unset. Otherwise, the value of the `in1` static code is set to the `x` literal.

Functionality: The `rel` statement describes input-output relation implemented at the output port `z` of the multiplier block under each mode. For example, when the block mode matches `(x,h,h)` or `(x,m,m)` the block multiplies the analog input signal `x` by the digital parameter `c` so that the expression implemented by `z` is `c*x`.

The block specification also includes a delta model specification that captures the space of behavioral deviations in the fabricated multiplier blocks. The multiplier block has two correctable delta model parameters (`u`, `v`) and one uncorrectable delta model parameter (`w`). The delta model parameter `w` implements an uncorrectable offset and would ideally take on a value close to zero. The delta model parameters `u` and `v` can be compensated for by the compiler. The `u` parameter implements an unexpected gain is compensated for by the `LScale` compilation pass, and the `v` parameter implements an unexpected data field offset and is compensated for by the HCDCv2 runtime.

The current ranges accepted at multiplier ports `x`,`y`, and `z` are defined with the `interval` statements on lines 23-25. Each port's value must fall within the port's operating range for the block to function correctly. The `interval` statements in Figure 5-6 define the operating ranges for ports `z`, `x`, and `y` as a function of the block mode. For example, the value of port `z` must remain between `-2` and `2` μA for odes matching `(*,*,m)` and between `-20` and `20` μA for modes matching `(*,*,h)`. The range of digital values supported by data field `c` is defined by the `interval` statement on line 22.

The noise introduced into port `z` defined with the `noise` statement on line 2. Each of these `noise` statements identifies the standard deviation of the noise associated with each block port. The block defined in Figure 5-6, for example, produces a noisy signal at port `z` – this signal has a standard deviation of `0.02` μA for modes matching `(* * m)` and a standard deviation of `0.2` μA for modes matching `(* * h)`.

```
block fan type assemble modes [(+,+,+,m),(-,+,+,m),(+,-,+,m),(+,+,-,m),(-,-,+,m),
    (-,+,-,m),(+,-,-,m)(-,-,-,m), (+,+,+,h),(-,+,+,h),(+,-,+,h),(+,+,-,h),(-,-,+,h),
    (-,+,-,h),(+,-,-,h)(-,-,-,h)] {
   in x analog current; out z0,z1,z2 analog current;

   rel z0 = func |(+,*,*,*) -> x |(-,*,*,*) -> -x
   rel z1 = func |(*,+,*,*) -> x |(*,-,*,*) -> -x
   rel z2 = func |(*,*,+,*) -> x |(*,*,-,*) -> -x


   delta-par a0,a1,a2 gain ideally 1.0;
   delta-par b0,b1,b2 offset ideally 0.0;
   delta z0 = func |(+,*,*,*) -> a0*x+b0 |(-,*,*,*) -> -a0*x+b0
   delta z1 = func |(+,*,*,*) -> a1*x+b1 |(-,*,*,*) -> -a1*x+b1
   delta z2 = func |(+,*,*,*) -> a2*x+b2 |(-,*,*,*) -> -a2*x+b2

   noise x,z0,z1,z2 = func | (*,*,*,m) -> 0.02 | (*,*,*,h) -> 0.04
   interval x,z0,z1,z2 = func |(*,*,*,m) -> [-2,2] |(*,*,*,h) -> [-20,20]; }
```

Figure 5-7: Block specification for current copier.

The multiplier imposes a maximum frequency limitation on the computation if the multiplier is configured to multiply `x` and `y`. The HCDCv2 cannot execute the mapped computation at a rate faster than 40 kHz when the multiplier is configured to multiply two time-varying signals. This restriction is necessary because the multiplier begins to exhibit unwanted frequency-gain characteristics at frequencies above 40 kHz.

Digital ports are quantized into a finite set of values as specified by their corresponding `quantize` and `interval` statements. The block defined in Figure 5-6, for example, quantizes `c` into `256` digital values between `-1` and `0.9921` (with the values spaced 0.0071825 apart). The `c` data field field uses the same value encoding scheme as the digital signals presented at the beginning of Section 5.7.

Calibration Codes: The `mul` block also works with 6 calibration codes (`bias0`,`bias1`, `nmos`, `pmos`, `biasOut`, and `gainCal`). The `bias0`, `bias1`, `biasOut`, and `gainCal` codes each take on values between 0 and 63. The `pmos` and `nmos` codes take on values between 0 and 7. These calibration codes are used to eliminate bias in the block and tune the block to achieve unity gain.

Fanout (`fan`) Block

The HCDCv2 provides a current copier (`fan`) block that copies analog currents so that

a particular signal may be used in more than one place. This block is integral to performing current-mode analog computation because analog currents cannot be used more than once without altering the signal. The `fan` block accepts an analog current input at port `x` and produces three analog current outputs at ports `z0`,`z1`,`z2`. It has four static codes (`sgn0`,`sgn1`,`sgn2`,`in`). The static codes `sgn0`,`sgn1`, and `sgn2` are either + or - and static code `in` is either `m` or `h`.

Figure 5-7 presents the block specification for the `fan` block. The block definition instantiates `fan` as an `assembly` block with 13 modes. The block accepts an analog current `x` as input and produces three analog currents `z0`, `z1`, and `z3` as outputs.

Modes: The defined modes encode all possible combinations of static code values for the `fan`. Each combination of static codes is encoded as a tuple of literals `(sgn,0,sgn1,sgn2,in)` where each tuple corresponds to a block mode.

Functionality:The block generally produces copies of the current at input port `x`. When `sgn0` is `-`, the first output is negated, when `sgn1` is `-`, the second output is negated, and when `sgn2` is `-`, the third output is negated. The `in` static code controls the current range accepted at input port `x`. When in `m` mode, port `x` accepts currents between [-2,2] μA. When in `h` mode, port `x` accepts currents between [-20,20] μA. The `in` static code affects the noise characteristics of the block – when in `h` mode, all signals are subject to more noise.

The block specification also provides a delta model for each output port. It defines six uncorrectable delta model parameters `a0`, `a1`,`a2`,`b0`,`b1`,`b2` which linearly transform the current produced at each output. The `a1`, `a1`, and `a2` parameters are technically correctable at compile time but make it more difficult for the compiler produce valid ADPs. For this reason they are marked uncorrectable. Ideally the `a0`, `a1`, and `a2` parameter values would be close to one and the `b0`, `b1`, and `b2` parameter values would be close to zero.

Calibration Codes: The `fan` block works with (`pmos`,`nmos`,`bias0`,`bias1`,`bias2`,`biasIn`). The `pmos` and `nmos` codes take on values between 0 and 7. The `bias0`, `bias1`, `bias2`, and `biasIn` codes take on values between 0 and 64. These codes are used to eliminate any bias in the block.

Digital-to-Analog Converter (`dac`) Block

The `dac` block converts a digital signal from either memory or a lookup table to an analog current (port `z`). The `dac` block is hardwired to two lookup tables which emit time-varying

```
block dac type compute modes [(const,m),(const,h),(dyn,m),(dyn,h)] {
   in x digital; out z analog current;
   data c const;
   rel z = func |(const,m) -> 2*c
        |(const,h) -> 20*c | (dyn,m) -> 2*x
        |(dyn,h) -> 20*x

   delta-par a correctable gain ideally 1;
   delta-par b correctable offset ideally 0;
   delta z = func |(const,m) -> 2*(a*c+b)
        |(const,h) -> 20*(a*c+b) | (dyn,m) -> 2*(a*x+b)
        |(dyn,h) -> 20*(a*x+b)

   quantize x,c = linear 256;
   noise z = func | (*,m) -> 0.01 | (*,h) -> 0.1
   interval x,c = [-1,0.998];
   interval z = func |(*,m) -> [-2,2] |(*,h) -> [-20,20]; }
```

Figure 5-8: Block specification for `dac`

digital signals. It has one dynamic code (`c`) and three static codes (`sgn`,`in`,`src`). Static code `in` may be either `m` or `h`, static code `sgn` may be either `+` or `-`, and static code `src` may be `mem`,`lut0`, or `lut1`. The `dac` routes the digital signal from the first lookup table when `src` is `lut0` and routes the digital signal from the second lookup table when `src` is `lut1`. The `sgn` code negates the produced analog signal when it is set to `-`. Because this functionality is not useful to the compiler, the HCDCv2 runtime fixes the `sgn` code to `+` and does not expose the behavior of this code in the ADS.

Figure 5-8 presents the ADS block specification for the digital-to-analog converter. The `dac` block is a `compute` block which accepts a digital input `x` and constant data field `c` and produces an analog output `z`. The data field `c` is mapped to the `dac` dynamic code `c` by the HCDCv2 runtime.

Modes:The `dac` definition specifies four modes which indirectly encode the values of the `in`, and `src` static codes as a tuple of literals:

if `src` = `mem` **then** `(const,in)` **else** `(dyn,in)`

The above encoding directly encodes the value of `src` in the tuple of literals. The HCDCv2 runtime automatically derives the final value of the `src` static code when the mode is `(dyn,in)` by analyzing the connections in the ADP.

Functionality:The `rel` statement describes the ideal input-output relation implemented at port `z`. Generally speaking, the block converts a digital signal to an analog current. The

```
block lut type compute modes [(*)] {
   in x digital; out z digital;
   data f expr vars +bracc;
   rel z =  call(f,[x])
   quantize x,z = linear 256;
   interval x,z = [-1,0.998];
}
```

Figure 5-9: Block specification for `lut`

`src` code determines whether the `dac` block converts the digital signal provided by one of the time-varying hardwired inputs (`x`) or if it converts the data field `c` to an analog signal. The `in` static code determines the magnitude of the produced analog signal at `z`.

The `delta` model for the `dac` block introduces the correctable `a` and `b` parameters. These parameters linearly transform the produced analog signal. The `a` parameter is compensated for by the `LScale` pass of the compiler and the `b` parameter is compensated for by the HCDCv2 runtime.

The block mode determines the current range of the analog signal at port `z`. The analog current at `z` is between [-2,2] μA when `in` is `m`. The analog current at `z` is between [-20,20] μA when `in` is `h`.

Calibration Codes: The `dac` block also works with three calibration codes (`pmos`,`nmos`,`gainCal`). The `nmos` and `pmos` calibration codes take on values between 0 and 7. The `gainCal` code takes on values between 0 and 64. These calibration codes are used to eliminate bias and tune the block to achieve unity gain.

LUT (`lut`) Block

The `lut` block applies a user-defined one-input/one-output function to a continuously evolving digital signal. The `lut` input is hardwired to two `adc` blocks which emit digital signals. The `lut` output is hardwired to two `dac` blocks. The `src` static code of the `dac` block determines which `lut` it reads from. The `lut` block one static code `src`) which determines which of the two `adc` blocks to read from. The `src` field may be either `adc0` or `adc1`. Each `lut` block also has an indexable memory segment containing 255 dynamic codes (`c[0]`...`c[255]`) which capture the behavior of the user defined function. At runtime, the `lut` block looks up the incoming digital value in its lookup table (the indexable segment of dynamic codes) and returns the result.

```
block adc type compute modes [(m),(h)] {
   in x analog current; out z digital;
   rel z = func |(m) -> 0.5*x | (h) -> 0.05*x

   delta-par a correctable gain ideally 1;
   delta-par b correctable offset ideally 0;
   delta z = func |(m) -> 0.5*(a*x+b) |(h) -> 0.05*(a*x+b)
   quantize z = linear 256;
   interval z = [-1,0.998];
   interval x = func |(m) -> [-2,2] |(h) -> [-20,20]
   noise x = func |(m) -> 0.01 |(h) -> 0.1
}
```

Figure 5-10: Block specification for `adc`

Figure 5-9 presents the ADS block specification for the `lut` block. The `lut` is a `compute` block with one mode. The HCDCv2 runtime automatically infers the value of the `src` static code by analyzing the connections in the ADP, so it does not need to be exposed to the compiler. The `lut` works with one digital input `x` and one digital output `z` and accepts an expression data field `f`. The expression data field `f` accepts one input variable and is mapped to a symbolic expression by the compiler. This symbolic expression is used to instantiate the 255 dynamic codes in the `lut` block.

Because the `lut` block operates entirely in the digital domain, it is not subject to the effects of noise and does exhibit any behavioral deviations which need to be captured by delta models. The `lut` block also does not need to be calibrated since it does not leverage any analog behavior. Both the digital input and output ports use the encoding scheme for digital signals presented at the beginning of Section 5.7.

Analog-to-Digital Converter (`adc`) Block

Figure 5-12 presents the ADS specification of the HCDCv2 analog to digital converter (`adc`) block. The `adc` block accepts an analog current input at port `x` and produces continuously evolving digital output at port `z`. This block is used in conjunction with an `dac` and `lut` blocks to implement an arbitrary time-varying one-input/one-output functions. It has one static code (`in`) which may either be assigned to m or h. The ADP block specification defines two modes which directly encode the value of this static code.

Functionality: Generally, the `adc` scales down and converts the analog value at port `x` to an digital signal between `[-1,1]`. The block works with currents between `[-2,2]` μA when

```
block extout type compute modes [(*)] {
   in x analog current; out z analog voltage;
   rel z = emit(0.6*x)
   interval z = [-2.0,2.0];
   interval z = [-1.2,1.2];
   noise z = 0.01; }

block extin type compute modes [(*)] {
   in x analog voltage; out z analog current;
   rel z = extvar(2.0*x)
   interval z = [-1.0,1.0];
   interval z = [-2.0,2.0];
   noise z = 0.01; }
```

Figure 5-11: Block specification for externally accessible ports (`extin` and `extout`)

in `(m)` mode and currents between `[-20,20]` μA when in `(h)` when in `(h)` mode. The block scales the incoming analog signal by a mode-dependent constant coefficient to normalize the signal. The analog signal at `x` is also subject to more noise when the block is in `(h)` mode. The time-varying digital values at `z` use the encoding scheme for digital signals presented at the beginning of Section 5.7.

The `adc` block specification also defines a delta model that describes the allowed behavioral deviations for the block. It defines two correctable delta model parameters `a` and `b` which linearly transform the incoming analog signal. The `a` delta model parameter is compensated for by the `LScale` scaling procedure. The `b` parameter is compensated for by the HCDCv2 runtime.

Calibration Codes: The `adc` block also works with 8 calibration codes: `pmos`,`pmos2`,`nmos`,`i2v_cal`,`upper_fs`,`upper`, `lower_fs`,`lower`. The `pmos`, `pmos2`, and `nmos` calibration codes take on values between 0 and 7. The `upper_fs` and `lower_fs` calibration codes take on values between 0 and 3. The `lower` and `upper` calibration codes take on values between 0 and 63. These calibration codes are used to eliminate bias and tune the block to achieve unity gain.

The `extin` and `extout` blocks

The `extout` block converts a current input (port `x`) into a voltage and routes the voltage signal to an externally accessible pin. This outgoing signal can be probed with measurement equipment or routed to peripheral circuits. This block has no static, dynamic, or calibration codes and has exactly one mode. It implements the function `0.6*x`, where `x` accepts analog

```
block cin type route modes [(*)] {
   in x analog current; out z analog current;
   rel z = x
   interval z = [-20.0,20.0]; }
block cout type route modes [(*)] {
   in x analog current; out z analog current;
   rel z = x
   interval z = [-20.0,20.0]; }
block tin type route modes [(*)] {
   in x analog current; out z analog current;
   rel z = x
   interval z = [-20.0,20.0]; }
block tout type route modes [(*)] {
   in x analog current; out z analog current;
   rel z = x
   interval z = [-20.0,20.0]; }

```

Figure 5-12: Block specification for routing blocks (`tin`, `tout`, `cin`, and `cout`)

currents between [-2,2] μA. The voltage emitted at `z` is between [-1.2,1.2] volts.

The `extin` block converts an externally provided analog voltage to an analog current which is internally accessible within the HCDCv2. The incoming signal can be supplied from an external peripheral device, such as a sensor. This block has no static, dynamic, or calibration codes and has exactly one mode. It implements the function `2.0*x`, where `x` accepts voltages between [-1,1] volts. The current emitted at `z` is between [-2,2] μA.

The Routing Blocks (`tin`,`tout`,`cin`, and `cout`)

The `tin`, `tout`, `cin`, and `cout` blocks are all used to route signals between substructures in the HCDCv2. They all accept one input at port `x` and produce one output at port (`z`). Each of the `route` blocks have exactly one mode and no static, dynamic, or calibration codes. These routing blocks all implement equality relations and are used to forward signals through the device.

Addition with Kirchoff's Law

The HCDCv2 does not offer blocks that perform addition and subtraction. Instead, these computations are performed by leveraging *Kirchoff's law*. Kirchoff's law states that the sum of all currents flowing into the same join point is zero. Two analog currents are therefore added together by connecting them to the same port. The signal flowing out of the port

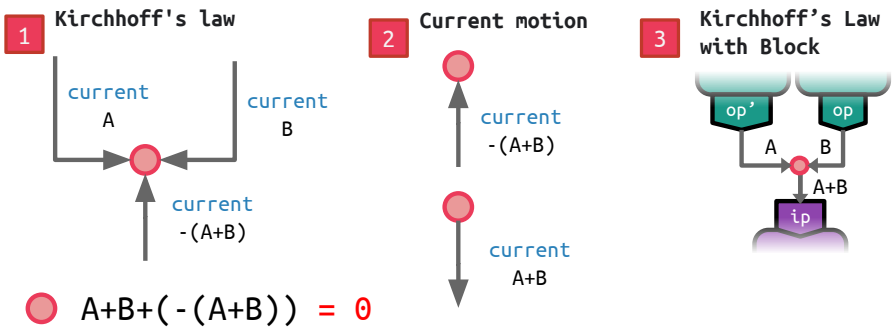


Figure 5-13: Overview of Kirchhoff's law

and into the block equals the sum of the signals flowing into that port.

Figure 5-13 graphically depicts how Kirchoff's law implements addition. If two wires carrying currents A and B flow into the same join point, then the current flowing out of the join point (wire C) equals A+B.

```
device hcdc {
 freq 126000;
 views chip,tile,slice,index;
 loc 0,1 chip; loc 0,1,2,3 in tile;
 loc 0,1,2,3 in slice; loc 0,1,2,3 index;
 blk tin, tout @ (*,*,*,*)
 blk int,mul,fan,dac,cin,cout @ (*,*,*,0)
 blk mul,fan @ (*,*,*,1)
 blk adc,lut @ (*,*,0,0)
 blk adc,lut @ (*,*,2,0)
 blk extin,extout @ (*,3,2,0)
 blk extin,extout @ (*,3,3,0)

 // block-to-block connections
 conn mul,dac,int,tin,fan @ (0,0,*,*) with mul,adc,int,tout,fan @ (0,0,*,*)
 conn lut @ (0,0,*,*) with dac @ (0,0,*,*)
 conn adc @ (0,0,*,*) with lut @ (0,0,*,*)

 conn mul,dac,int,tin,fan @ (0,1,*,*) with mul,adc,int,tout,fan @ (0,1,*,*)
 conn lut @ (0,1,*,*) with dac @ (0,1,*,*)
 conn adc @ (0,1,*,*) with lut @ (0,1,*,*)

 conn mul,dac,int,tin,fan @ (0,2,*,*) with mul,adc,int,tout,fan @ (0,2,*,*)
 conn lut @ (0,2,*,*) with dac @ (0,2,*,*)
 conn adc @ (0,2,*,*) with lut @ (0,2,*,*)

 conn mul,dac,int,tin,fan @ (0,3,*,*) with mul,adc,int,tout,fan @ (0,3,*,*)
 conn lut @ (0,3,*,*) with dac @ (0,3,*,*)
 conn adc @ (0,3,*,*) with lut @ (0,3,*,*)

  conn mul,dac,int,tin,fan @ (1,0,*,*) with mul,adc,int,tout,fan @ (1,0,*,*)
 conn lut @ (1,0,*,*) with dac @ (1,0,*,*)
 conn adc @ (1,0,*,*) with lut @ (1,0,*,*)

 conn mul,dac,int,tin,fan @ (0,1,*,*) with mul,adc,int,tout,fan @ (0,1,*,*)
 conn lut @ (1,1,*,*) with dac @ (1,1,*,*)
 conn adc @ (1,1,*,*) with lut @ (1,1,*,*)

 conn mul,dac,int,tin,fan @ (0,2,*,*) with mul,adc,int,tout,fan @ (0,2,*,*)
 conn lut @ (1,2,*,*) with dac @ (1,2,*,*)
 conn adc @ (1,2,*,*) with lut @ (1,2,*,*)

 conn mul,dac,int,tin,fan @ (0,3,*,*) with mul,adc,int,tout,fan @ (0,3,*,*)
 conn lut @ (1,3,*,*) with dac @ (1,3,*,*)
 conn adc @ (1,3,*,*) with lut @ (1,3,*,*)
```

Figure 5-14: HCDCv2 device specification

```
  // inter-tile connections
  conn tout @ (0,*,*,*) with tin @ (0,*,*,*)
  conn tout @ (1,*,*,*) with tin @ (1,*,*,*)

  // tile-chip and tile-external input/output connections
  conn tout @ (0,0,*,*) with cout,extout @ (0,0,*,*)
  conn tout @ (0,1,*,*) with cout,extout @ (0,1,*,*)
  conn tout @ (0,2,*,*) with cout,extout @ (0,2,*,*)
  conn tout @ (0,3,*,*) with cout,extout @ (0,3,*,*)
  conn tout @ (1,0,*,*) with cout,extout @ (1,0,*,*)
  conn tout @ (1,1,*,*) with cout,extout @ (1,1,*,*)
  conn tout @ (1,2,*,*) with cout,extout @ (1,2,*,*)
  conn tout @ (1,3,*,*) with cout,extout @ (1,3,*,*)

  conn extin,cin @ (0,0,*,*) with tin @ (0,0,*,*)
  conn extin,cin @ (0,1,*,*) with tin @ (0,1,*,*)
  conn extin,cin @ (0,2,*,*) with tin @ (0,2,*,*)
  conn extin,cin @ (0,3,*,*) with tin @ (0,3,*,*)
  conn extin,cin @ (1,0,*,*) with tin @ (1,0,*,*)
  conn extin,cin @ (1,1,*,*) with tin @ (1,1,*,*)
  conn extin,cin @ (1,2,*,*) with tin @ (1,2,*,*)
  conn extin,cin @ (1,3,*,*) with tin @ (1,3,*,*)

  // inter-chip connections
  conn cout @ (0,0,0,0) with cin @ (1,1,3,0)
  conn cout @ (0,0,1,0) with cin @ (1,1,2,0)
  conn cout @ (0,0,2,0) with cin @ (1,1,1,0)
  conn cout @ (0,0,3,0) with cin @ (1,1,0,0)
  conn cout @ (0,2,0,0) with cin @ (1,0,3,0)
  conn cout @ (0,2,1,0) with cin @ (1,0,2,0)
  conn cout @ (0,2,2,0) with cin @ (1,0,1,0)
  conn cout @ (0,2,3,0) with cin @ (1,0,0,0)
  conn cout @ (0,3,0,0) with cin @ (1,3,0,0)
  conn cout @ (0,3,1,0) with cin @ (1,3,1,0)

  conn cout @ (1,0,0,0) with cin @ (0,1,3,0)
  conn cout @ (1,0,1,0) with cin @ (0,1,2,0)
  conn cout @ (1,0,2,0) with cin @ (0,1,1,0)
  conn cout @ (1,0,3,0) with cin @ (0,1,0,0)
  conn cout @ (1,2,0,0) with cin @ (0,0,3,0)
  conn cout @ (1,2,1,0) with cin @ (0,0,2,0)
  conn cout @ (1,2,2,0) with cin @ (0,0,1,0)
  conn cout @ (1,2,3,0) with cin @ (0,0,0,0)
  conn cout @ (1,3,0,0) with cin @ (0,3,0,0)
  conn cout @ (1,3,1,0) with cin @ (0,3,1,0)
}
```

Figure 5-15: HCDCv2 device specification (continued)

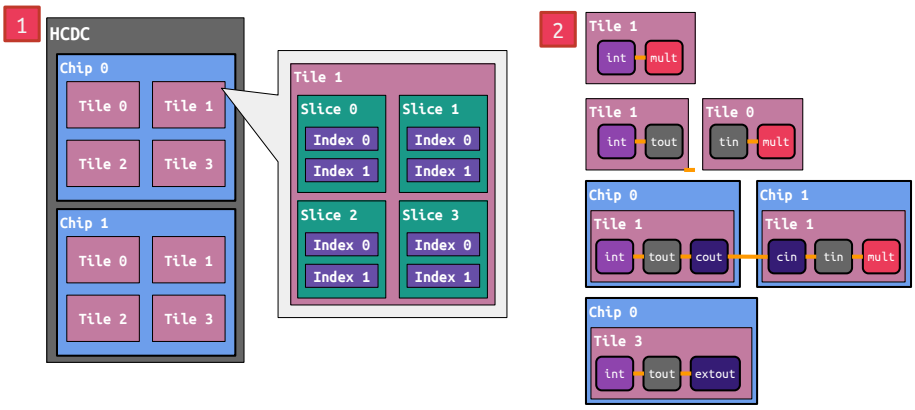


Figure 5-16: HCDCv2 chip, tile, slice, and index layout and connection rules.

5.7.2 HCDCv2 Layout

The HCDCv2 has between 16-128 hierarchically organized instances of each block. The HCDCv2 has two chips, where each chip has four tiles, each tile has four slices, and each slice has four indices. Subfigure 1 of Figure 5-16 presents a diagram of how the chips, tiles, slices, and indices are laid out on the HCDCv2. Block instances are uniquely identified by the chip, tile, slice, and index they reside at. Each HCDCv2 index may therefore contain no more than one instance of each block type. The HCDCv2 has a `mul` and `fan` blocks at indices 0 and 1 of each slice, `int` and `dac` blocks at index 0 of each slice, and `adc` and `lut` blocks at index 0 of even slices. The HCDCv2 has `tin` and `tout` blocks at each index and `cin` and `cout` blocks at index 0 of each slice. Each chip has two `extin` and two `extout` blocks which are at chip 0, tile 3, slice 2, index 0 and chip 0, tile 3, slice 3, index 0.

The HCDCv2 provides a collection of digitally programmable connections which may be enabled on the fly. Generally, blocks that are physically close together on the HCDCv2 are easier to connect together than blocks that are far apart. The HCDCv2 requires distant connections to be indirectly made by routing the signal through several different routing blocks. Subfigure 2 of Figure 5-16 presents an overview of how blocks colocated on the same tile, on different tiles, and on different chips may be connected together. All blocks within the same tile (except `extin` and `extout` blocks) may be connected together by enabling a single programmable connection. Blocks on different tiles may be connected together by routing the signals through a `tout` block on the source block's tile and then through a `tin` block on the destination block's tile. Blocks on different chips may be connected together by routing signals first through `tout`, `cout`, `cin`,`tin`, and `cin` blocks. Any block can be

connected to a `cext` block on the same tile provided the signal is routed through a `tout` block.

Figure 5-14 and Figure 5-15 presents the partial device specification for the HCDCv2. I omit the connection statements from the above specification for brevity. The HCDCv2 block locations are made up of four integer addresses. The values correspond to the chip, tile, slice, and index of the block. The HCDCv2 device specification describes four views: chip, tile, slice, and index. Lines 3-4 describe a device layout with two chips, four tiles per chip (numbered 0-3), four slices per tile (numbered 0-3), and four indices per slice (numbered 0-3).

Lines 6-12 define the block instances available on the HCDCv2 device. The HCDCv2 contains `tin` and `tout` routing block instances at every index on the device. The HCDCv2 contains instances of the `mul` and `fan` blocks at indices 0 and 1 of every slice and instances of `int` and `dac` blocks at index 0 of every slice. The `adc` and `lut` blocks are only available at index 0 of even slices. The `extin` and `extout` blocks are only available at two locations per chip.

The remainder of the device specification defines connections between block instances on the device. Lines 25-45 of Figure 5-14 defines the programmable connections between the compute and assembly blocks that reside on the same tile. Lines 2 and 3 of Figure 5-15 defines the programmable connections between tiles residing on the same chip. Lines 6-13 of Figure 5-15 defines the programmable connections between chip/external inputs and tile inputs. Lines 15-22 of Figure 5-15 defines the programmable connections between tile outputs and chip/external outputs. Lines 25-45 of Figure 5-15 define the connections between chips.

5.8 HCDCv2 Manufacturing Variations, Calibration, and Delta Models

Due to variations in the manufacturing process, individual HCDCv2 blocks experience variations in behavior in practice. These behavioral variations change depending on the block mode and how the block is calibrated. The HCDCv2 implements calibration strategies that eliminate unwanted behaviors from the hardware. After calibration, the HCDCv2 runtime

identifies the delta model parameters for each of the calibrated blocks. These delta model parameters are used by the compiler to more accurately target the device on hand. This section describes the calibration strategies deployed by the HCDCv2, the workflow for populating the delta model database. I also present a multiplier case study that explores the effect the two calibration strategies have on the delta models for a multiplier block.

5.8.1 HCDCv2 Calibration

The HCDCv2 firmware implements multiple user-settable calibration strategies that can be used to calibrate the HCDCv2 blocks. These calibration strategies identify the best calibration code values for a block instance under a given block mode. The criteria for determining the best set of calibration code assignments depends on the calibration strategy used:

- **Traditional Calibration Strategy** (`minimize_error`): The HCDCv2 firmware implements a traditional calibration strategy that calibrates the blocks so that the delta model parameters adhere to their ideal values. This calibration strategy effectively calibrates the blocks to implement the input-output relations defined by the block `rel` statements. The `minimize_error` calibration strategy is the calibration strategy traditionally used in hardware design.
- **Co-Designed Delta Model** (`maximize_fit`): The HCDCv2 firmware implements a co-designed calibration strategy that prioritizes eliminating hard-to-correct unwanted behaviors such as point errors and unwanted biases in the calibrated blocks. I designed this calibration strategy with knowledge of what the compiler can statically compensate for in mind. This calibration strategy allows for behavioral deviations that the compiler can compensate for in compilation. With this calibration strategy, the blocks are calibrated such that the uncorrectable delta model parameters adhere to their ideal values. The correctable delta model parameters are allowed to deviate from their ideal values under this strategy. The `maximize_fit` calibration is a novel compilation-aware calibration strategy that allows for controlled behavioral deviations, provided these deviations can be compensated for in software.

The device firmware executes the HCDCv2 calibration procedure. Each block calibration strategy performs a heuristic search that efficiently searches over calibration code values

to find the best set of calibration codes. The firmware evaluates each set of candidate calibration code values by testing the block instance on a small set of test points. The calibration code values returned by the calibration procedure are written to the calibration database. The HCDCv2 runtime retrieves these calibration code values and writes them to the HCDCv2 before executing an ADP.

Uniqueness: Each set of calibration code values is uniquely identified by the block instance name and location, the calibration strategy used to derive the values, and a block mode. The static code values encode the mode of the block.

5.8.2 HCDCv2 Delta Models

The HCDCv2 analog device specification defines mode-dependent delta model specifications for each of the blocks. The HCDCv2 runtime elicits delta model parameter values from the device. Each set of delta model parameters is uniquely identified by the block instance name and location, the output port name, the calibration strategy used to derive the values, and the block mode. The static codes of the block encode the block mode. The compiler uses both the delta model specifications from the ADS and the delta model parameters from the delta model database to effectively target the device at hand. The delta model specification can be concretized to implement the following relations:

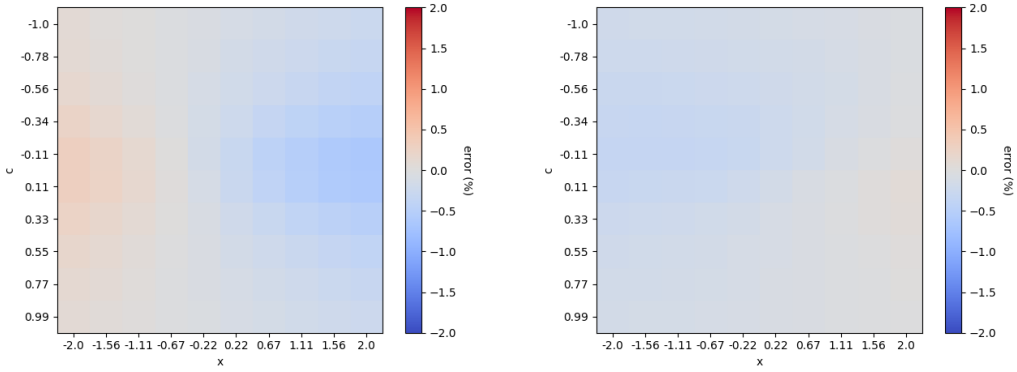
- **Ideal Input-Output Relation**: The ideal input-output relation describes the behavior of the block if no behavioral variations occur. The delta model specification implements the ideal input-output relation of a block instance when all delta model parameters are set to their ideal values. Under these conditions, the delta model specifications match the ideal input-output relations (`rel` statements) defined with block. The traditional `minimize_error` calibration strategy calibrates the block to implement the ideal input-output relation of the block.

- **Delta Model**: The delta model captures the empirically observed behavioral deviations of the block. The delta model specification captures the empirical behavior of a block instance when all of the delta model parameters are set to the delta model parameter values from the delta model database. The delta model captures the subset of behavioral deviations that delta model specification can model effectively.

- **Correctable Delta Model**: The compiler works with the correctable delta model that captures the subset of behavioral deviations that can be compensated for in compilation. The delta model specification captures the correctable behavior of a block instance when all of the uncorrectable delta model parameters are set to their ideal values, and all of the correctable parameters are set to the empirically derived delta model parameters from the delta model database. The co-designed `maximize_fit` calibration strategy calibrates the block so that only the correctable delta model parameters deviate from their ideal values.

The HCDCv2 runtime identifies the delta model parameters for the blocks by profiling the blocks offline. The collected data is then fit to the block delta model specifications from the ADS to derive the delta model parameter values for each configured block instance. These parameters are stored in a delta model database. The compiler retrieves parameter values from this database to derive the correctable input-output relations for the relevant block instances. Refer to Sections 5.9.1-5.9.3 for more information on the profiling and model elicitation process.

5.8.3 Example: `mul` block



(a) `minimize_error` (b) `maximize_fit`

Figure 5-17: Block errors associated with the `minimize_error` and `maximize_fit` calibration strategies for multiplier `(1,3,0,0)`.

This case study investigates the ideal behavior, and the correctable and uncorrectable behavioral deviations present in a calibrated multiplier block in `(x,m,h)` mode. When the

mul block is in (x,m,h) mode, the static codes in0,in1 are m, the static code out is h, and the vga static code is true. I present the delta model specification for the multiplier block below:

```
1    delta-par u,v correctable;
2    delta-par w;
3    delta z = func |(m,m,h) -> u*5*x*y+w,
4         |(m,m,m)|(h,m,h)|(m,h,h) -> u*0.5*x*y+w
5         |(h,m,m)|(m,h,m)|(h,h,h) -> u*0.05*x*y+w
6         |(x,m,h)->10*(u*c+v)*x+w |(x,h,m) -> 0.1*(u*c+v)*x+w
7         |(x,h,h)|(x,m,m) -> (u*c+v)*x+w;
```

In the above relation, u,v, and w delta model parameters all capture variations in gain and bias in the block. The ideal values for the delta model parameters u,v, and w are 1, 0, and 0 respectively. The u and v parameters are correctable. The delta model specification defines the input-output relation 10*(u*c+v)*x+w when the block is placed in (x,m,h) mode.

- **Ideal Input-Output Relation**: The mul block ideally implements 10*c*x when the block is in (x,m,h) mode. The delta model relation implements 10*(1*c+0)*x+0 or 10*c*x when all of the delta model parameters are set to their idealized values. This matches the expression defined by the rel statement for the block.

This case study investigates the behavior of the multiplier block at chip 1, tile 3, slice 3, index 0 of the HCDCv2 board at hand. I calibrate this block with both the minimize_error and maximize_fit calibration strategies to get the set of calibration code values. I then profile the block by exercising it over the space of values c and x may take on. For each combination of c and x, the output current at z is measured to obtain the empirically observed output. This profiling dataset is then fit to the above delta model with u, v, and w as free variables. The calibrated blocks are then profiled and analyzed to identify the delta model parameter values:

- **Minimize Error**: The error minimization calibration procedure assigns the calibration codes pmos, nmos, gainCal, bias0, bias1, and biasOut to 3, 2, 63, 34, 32, and 41 respectively. The calibrated block assigns the u, v, and w parameters to 0.979, 0.0346 , and -0.102 respectively.

- **Maximize Fit**: The delta model fit maximization calibration procedure assigns the calibration codes `pmos`, `nmos`, `gainCal`, `bias0`, `bias1`, and `biasOut` to 1, 1, 10, 33, 32, and 39 respectively. The calibrated block assigns the `u`, `v`, and `w` parameters to `0.736`, `0.0284`, and `0.0104` respectively.

The delta models for port `z` the multiplier at chip 1, tile 3, slice 0, index 0 of the HCDCv2 board when the multiplier is in `(x,m,h)` mode are as follows:

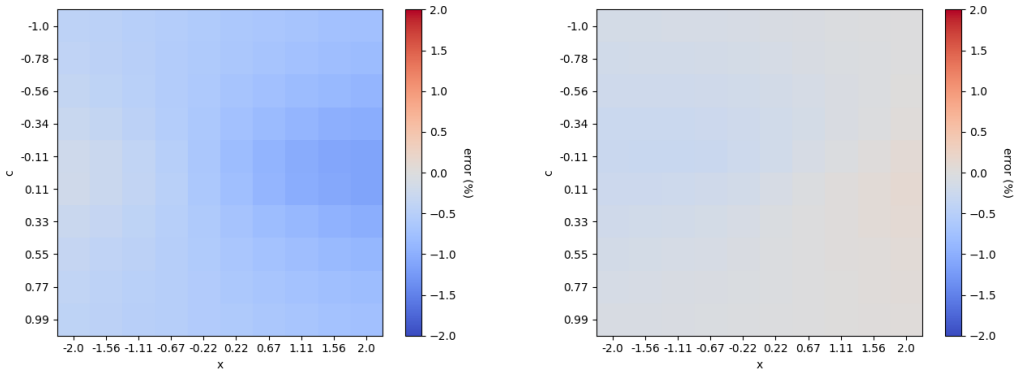
- **Minimize Error**: The delta model of output port `z` is `(0.979*c - 0.0346)*x-0.102` when the block is in `(x,m,h)` mode and has been calibrated with the `minimize_error` calibration strategy.

  While this behavior closely resembles the ideal input-output relation `c*x`, it does introduce some degree of uncorrectable error. The `w` term is `-0.102`, which is about 0.5% of the dynamic range of the signal. Figure 5-17a presents the input-dependent static error in the calibrated block that cannot be captured with the delta model. This static error is referred to as unmodellable since it cannot be captured with the delta model. This calibrated block exhibits between 0% and 1% unmodellable error depending on the value of `x` and `c`. This unmodellable error occurs because the calibration strategy cannot fully eliminate all unwanted behavior from the block.

- **Maximize Fit**: The delta model for port `z` is `(0.736*c-0.0284)*x + 0.0104` when the block is in `(x,m,h)` mode and has been calibrated with the `maximize_fit` calibration strategy.

  While this behavior deviates from the ideal input-output relation `10*c*x`, it introduces far less uncorrectable error into the behavior of the block. The `w` delta model parameter value is `0.0104`, which is about 0.05

The compiler can is only able to compensate for the correctable delta model parameters in the delta model specification. The delta model specification for output port `z` under `(x,m,m)` mode is `(u*c+v)*x+0` or `(u*x+v)*x` if only the correctable parameters are allowed to deviate from their ideal values (`w` is 0). The correctable input-output relations for the multiplier at chip 1, tile 3, slice 0, index 0 of the HCDCv2 board, when the multiplier is in `(x,m,h)` mode, are as follows:



(a) `minimize_error` (b) `maximize_fit`

Figure 5-18: Uncorrectable delta model error associated with the `minimize_error` and `maximize_fit` calibration strategy for multiplier `(1,3,0,0)`.

- **Minimize Error**: The correctable input-output relation of output port `z` is `(0.979*c - 0.0346)*x` when the block is in `(x,m,h)` mode and has been calibrated with the `minimize_error` calibration strategy. The `1.02` and `0.04` delta model parameter values can be statically compensated for in compilation. For example, to implement `0.6*x`, I would compute the solution to `0.979*c - 0.0346 = 0.6` to find the value of data field `c`. I find that `c` should be set to `0.648` instead of `0.6`. The offset (`-0.102`) cannot be compensated for and becomes part of the block's uncorrectable delta model error.

  Figure 5-18a presents the error at output port z if I use the correctable delta model for the calibrated block. This error is the discrepancy between the measured values and the values computed by the correctable input-output relation. When the block is calibrated with this calibration strategy, the compiler cannot compensate for a significant amount of the error present in the block.

- **Maximize Fit**: The correctable input-output relation for port `z` is `(0.736*c - 0.0284)*x` when the block is in `(x,m,m)` mode and has been calibrated with the `maximize_fit` calibration strategy. The `0.736` and `0.0284` delta model parameter values can be statically compensated for in compilation. For example, to implement `0.6*x`, I would compute the solution to `0.736*c - 0.0284 = 0.6` to find the value of data field `c`. I find that `c` should be set to `0.854` instead of `0.6`. The offset (`0.0104`) cannot be compensated for and becomes part of the block's uncorrectable delta model

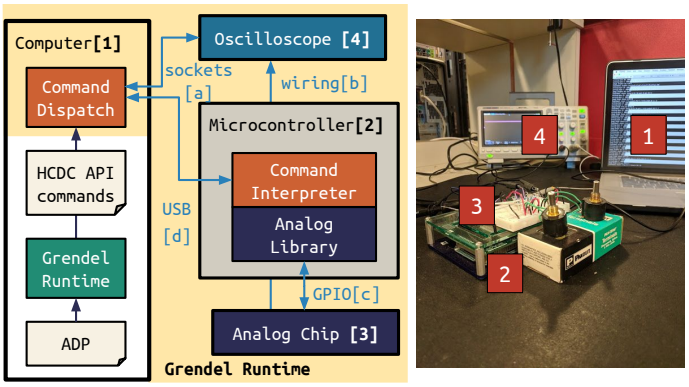


Figure 5-19: Laboratory Setup for HCDCv2

error.

Figure 5-18b presents the error at output port z if I use the correctable delta model for the calibrated block. This error is the discrepancy between the measured values and the values computed by the correctable input-output relation. When the block is calibrated with this calibration strategy, the compiler can compensate for a significant amount of the error present in the block.

5.9 HCDCv2 Software Stack and Runtime

The HCDCv2 runtime provides high-level operations for calibration, profiling, and model inference. These operations are used to calibrate the HCDCv2 and elicit delta models for the calibrated device. The HCDCv2 runtime extensively caches board information in several databases to avoid unnecessarily calibrating and characterizing blocks on the device. The HCDCv2 runtime also provides high-level operations for configuring blocks, setting connections, and running simulations. These operations are used to execute an input ADP on the target analog device.

Figure 5-19 presents an overview of the runtime environment. The HCDCv2 analog chip is hard-wired to an Arduino Due microcontroller running a command interpreter for the HCDCv2 API. This microcontroller runs all of the calibration and profiling logic and applies the dispatched HCDCv2 commands to the HCDCv2. The externally available pins on the HCDCv2 are connected to a networked Sigilent 1020XE oscilloscope. I use this measurement device to collect waveforms from the HCDCv2. Both the microcontroller and

the oscilloscope communicate with the host machine over USB and sockets, respectively.

The HCDCv2 runtime implements each high-level operation as a sequence of queries to the HCDCv2 API presented in Section 5.9.1. The HCDCv2 API is a low-level interface used for programming the HCDCv2 and any measurement devices. Each HCDCv2 API invocation is either dispatched to the command interpreter on the microcontroller or the oscilloscope. The microcontroller parses each HCDCv2 API command and configures the attached HCDCv2 accordingly. The oscilloscope HCDCv2 API calls are used to configure the oscilloscope. The runtime collects and processes any returned values from the oscilloscope and microcontroller.

5.9.1 HCDCv2 Low-Level Programming Interface

The HCDCv2 exposes an API for configuring analog blocks, enabling and disabling connections, and executing the simulation. This API also offers endpoints for profiling and calibration blocks on the chip. The block calibration procedure the calibration code values that eliminates the most unwanted block behavior under the provided set of static code assignments. The calibration procedure works with a calibration strategy that tells the firmware what constitutes unwanted behavior. The block profiling procedure tests a block with analog inputs and dynamic and static code assignments and returns the measured block output. I summarize the API endpoints below:

`set_state` **command**: The `set_state` command takes as input the kind of block to update, the chip, tile, slice, and index number of the target block, and the static, dynamic, and calibration code assignments to write to the block. The firmware writes the provided codes to the block at the specified location on the HCDCv2.

`enable_conn` **command**: The `enable_conn` command takes as input a source and destination port instance description. A port instance description consists of the port name, the kind of block the port belongs to, and the block instance's chip, tile, slice, and index number. The firmware turns on the connection between the specified source and destination ports.

`disable_conn` **command**: The `disable_conn` command takes as input a source and destination port description. The firmware turns off the connection between the specified source and destination ports.

`calibrate` **command**: The calibrate command takes as input the kind of block to calibrate and the chip, tile, slice, and index number of that block. It also takes in the calibration strategy the calibration procedure should use to evaluate calibration code assignments. The firmware calibrates the selected block with the provided calibration strategy. The calibration procedure uses the static code assignments that have previously been written to the target block.

`profile` **command**: The profile command takes as input the kind of block to profile, the output port to probe, and the chip, tile, slice, and index number of that block. It also takes, as input, one or more analog input values to feed into the target block. The firmware generates the required analog signals, routes these signals into the target block, and measures the signal at the desired output port. It returns the mean and standard deviation of the signal at the port. The profile command also supports specialized profiling operations for measuring the gain and bias of integrators – these specialized profiling operations are selected with an optional `method` argument.

`set_sim_time` **command**: The profile command takes as input the amount of time, in wall clock seconds, to execute the simulation for. The firmware caches the simulation time in memory for later use.

`run_sim` **command**: The simulation execution command takes no inputs and returns no outputs. The firmware powers on the programmed circuit and runs it for the simulation time stored in firmware memory.

oscilloscope commands: The API offers commands for configuring a networked Sigilent 1020XE oscilloscope. These commands support configuring the trigger signal, setting the voltage and time scale, and retrieving waveforms from the oscilloscope. The oscilloscope samples the voltage signal at the externally accessible output port `z` of the `cext` block at location `(0,3,2,0)`. All oscilloscope commands are directly dispatched to the measurement device.

5.9.2 The Calibration, Profiling, and Delta Model Databases

Because profiling and calibration are time-consuming operations, the HCDCv2 runtime environment internally caches profiling and calibration data for later use. This runtime environment tracks board-specific calibration and characterization information with a series of

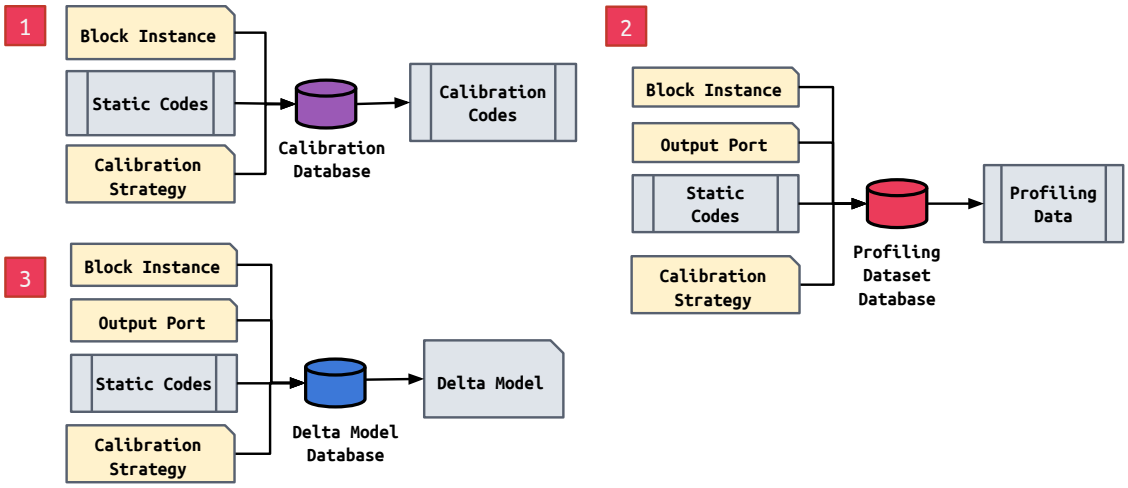


Figure 5-20: Overview of query structure to the HCDCv2 calibration, delta model, and profiling databases

databases. Figure 5-20 presents an overview of the HCDCv2 runtime databases:

- **Calibration Database**: The calibration database stores the calibration code assignments returned by the `calibrate` invocations. The block instance, static code assignments, and calibration strategy together uniquely identify each set of calibration code assignments in the calibration database. When the runtime calibrates a block, it first checks for an existing entry in the calibration database. If an entry already exists, the runtime returns the cached calibration code assignments.

  Profiling Dataset Database: The profiling dataset database stores the profiling data returned by the `profile` invocations. Each dataset contains one or more input-output tuples where inputs are both dynamic code assignments and analog inputs. The block instance, probed output port, static code assignments, and calibration strategy together uniquely identify each profiling dataset. The runtime uses the profiling datasets to derive the delta model parameters for each calibrated, configured block instance.

  Delta Model Database: The delta model database stores the delta model parameter values for each calibrated, configured block instance. The delta model parameters and the delta model specification together capture the empirically observed behavioral deviations at each output port of each block instance. The block instance, output port, static code assignments, and calibration strategy uniquely identify the delta

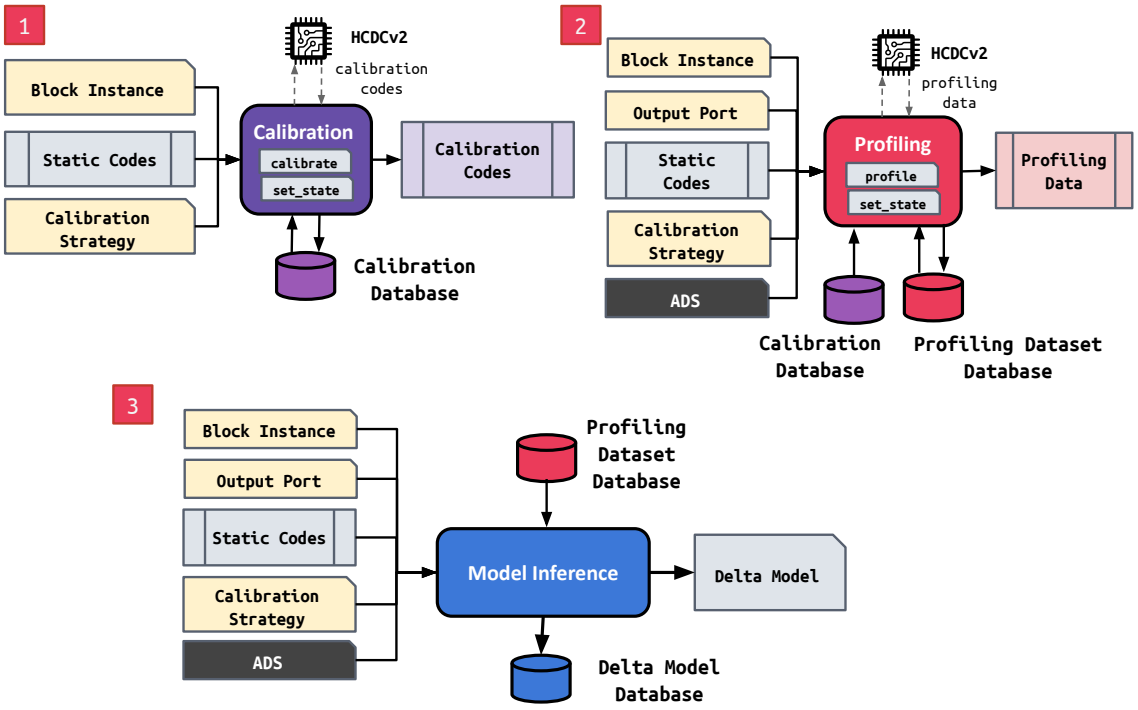


Figure 5-21: Overview of the calibration, profiling, and model inference operations.

model parameter values for each configured, calibrated output port instance. The compiler and runtime combine the delta model parameter values and the delta model specification to produce correctable delta models – these correctable delta models are used to statically compensate for behavioral deviations in software.

5.9.3 Calibration, Profiling, and Model Elicitation

Figure 5-21 presents the operation of the calibration, profiling, and delta model inference routines offered HCDCv2 runtime. These high-level operations cache calibration and profiling information to avoid unnecessarily reexecuting `calibrate` and `profile` commands. All of these routines work with a target block instance to calibrate or characterize, a set of static code assignments to write to the block, and a calibration strategy to use:

Calibration: The calibration routine calibrates the provided block instance using the specified calibration strategy under the provided set of static code assignments. It calibrates the block instance and stores the returned set of calibration code assignments in the calibration database. If the block instance has already been calibrated with this calibration strategy and set of static code assignments, it returns the calibration code assignments from the

database. To calibrate the block, it executes the `set_state` operation to write the static codes to the block instance and then executes the `calibrate` operation to calibrate the block instance with the provided calibration strategy. It writes the calibration code assignments to the calibration database.

Profiling: The profiling routine characterizes the behavior of the output port belonging to a calibrated block instance under the provided set of static codes. The profiling routine accepts, as input, the location and output port of the target block, the static codes, and the calibration strategy to use with the target block, and the analog device specification. The profiler reads calibration data from the calibration database and writes profiling data to the profiling dataset database. The profiling routine characterizes the block's behavior over the block's input space by exercising the block over a set of evenly spaced test inputs. The device ADS identifies the range of the test input values the profiler should generate. It returns the dataset of observed input-output pairs. The runtime records the data in the profiling dataset database and returns the dataset to the user.

The profile routine first looks up the calibration code assignments for the block instance under the provided set of static code assignments and calibration strategy in the calibration database. The runtime then writes the static and calibration code assignments to the provided block instance with the `set_state` command. After the block is configured, the profiler is ready to test the block. It breaks up the space of dynamic codes and analog inputs into a grid of evenly spaced points. The runtime dispatches a `profile` operation to the HCDCv2 for each combination of inputs. Each profile operation drives the provided inputs into the target block and returns the measured signal at the specified output port. The runtime collects these input-output pairs into a profiling dataset and then writes this dataset to the profiling database. If there's already an entry in the database, the runtime adds the collected data to the existing dataset. Note that the HCDCv2 runtime does not profile blocks that already have large datasets.

Model Inference: The model inference routine infers the delta model parameter values for the provided output port instance under the provided static codes and calibration strategy. The inference routine accepts, as input, the target block location and output port, the static code values and the calibration strategy to use with the block, and the analog device specification. The model inference routine first reads the profiling dataset for the configured, calibrated output port instance from the calibration database. The inference routine fits the

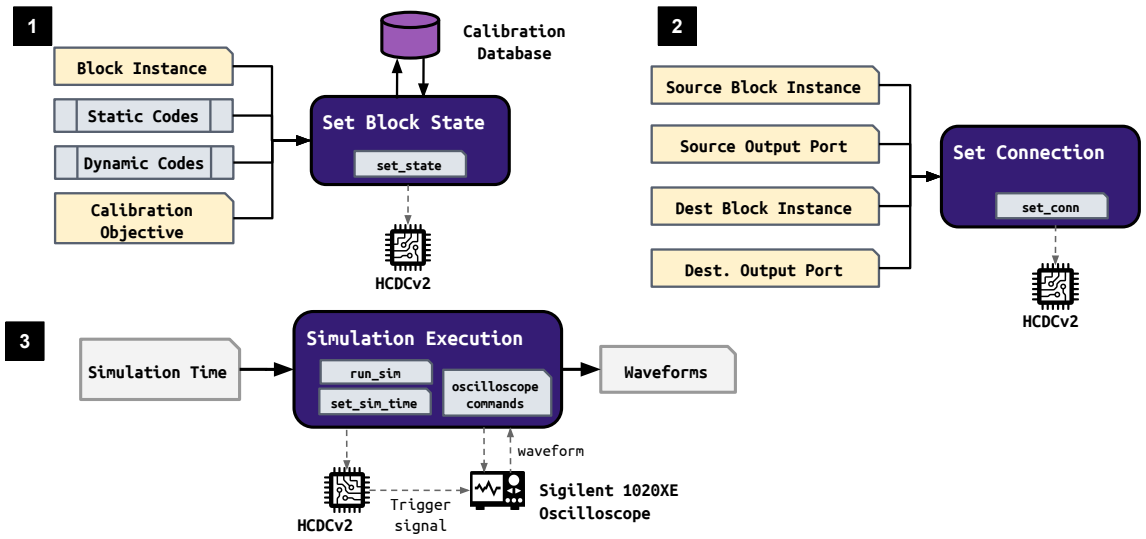


Figure 5-22: HCDCv2 block configuration, connection, and execution operations

profiling data to the delta model specification specified in the ADS and then writes the inferred delta model parameter values to the delta model database.

5.9.4 Analog Device Program Execution

Figure 5-22 presents the HCDCv2 runtime operations for configuring blocks, setting connections, and simulating circuits. These high-level operations automate some of the bookkeeping operations required to configure and execute a circuit.

- **Set Block State**: This operation writes a set of static, dynamic, and calibration code assignments to a specific block instance. This configuration operation accepts, as input, the target block instance, the set of static and dynamic code values to write to the block, and the calibration strategy to use for the block. The HCDCv2 runtime fills in the calibration code assignments by looking up the block instance, static codes, calibration strategy, and block instance in the calibration database. THe runtime then then writes the static, dynamic, and calibration code assignments to the target block instance with the `set_state` command.

- **Set Connection**: This operation enables a connection between the input and output port of two block instances. It accepts as input the source block instance and output port and the destination block instance and input port. It dispatches a single `set_-`

`conn` command which enables the desired connection.

- **Simulation Execution**: This operation runs the configured circuit on the HCDCv2 and returns the waveform. The runtime accepts as input the amount of time (in seconds) to execute the simulation for. The routine first dispatches the `set_sim_time` command to prepare the chip for running the desired experiment for the specified amount of time. The routine also configures the oscilloscope voltage and time division settings and sets up the oscilloscope to begin recording when a trigger signal is received. The routine then dispatches the `run_sim` command, which executes the circuit for the desired amount of wall-clock time. The HCDCv2 triggers the oscilloscope just before starting the computation. The oscilloscope returns the waveform of the measured external signal.

Together these operations write an analog circuit to the HCDCv2 and execute the analog circuit for a specified amount of wall clock time.

5.9.5 ADP Execution on the HCDCv2

The HCDCv2 runtime system supports executing an input ADP on the HCDCv2. The HCDCv2 runtime system accepts, as input, an ADP which targets the HCDCv2 analog device specification and the amount of time to simulate the dynamical system for (t).

The runtime system first writes the circuit described by the ADP to the HCDCv2. It translates each ADP `config` statement into a set of static and dynamic code assignments and writes the block state to the HCDCv2 with the set state runtime command:

- **Mode Translation**: The runtime looks up each block mode in an internal mode-static code look-up table to identify the correct set of static code assignments for the chosen block mode.

- **Constant Data Field Translation**: The runtime scales each ADP constant data field value by its associated scaling factor (`scale` statements) and then adjusts the scaled value to compensate for any correctable offsets in the block. The runtime retrieves the delta model parameter values for the configured, calibrated block instance from the delta model database. The computed decimal value is then translated into

a byte value using the `quantize` and `interval` statements from the ADS block specification.

- **Expression Data Field Translation**: The runtime adjusts each expression data field value to compensate for any correctable offsets in the attached `adc` and `dac` blocks. The runtime retrieves the delta model parameter values for the `adc` and `dac` blocks from the delta model database. The resulting modified expression is then translated to a look-up table using the `quantize` and `interval` statements from the block specification. These look-up table values are assigned to the appropriate dynamic codes.

The ADP `conn` statements are written to the HCDCv2 with the HCDCv2 runtime `set connection` routine. At this point, the ADP is fully programmed onto the HCDCv2 and is ready for execution. The HCDCv2 runtime computes the number of wall-clock seconds to record from the runtime in dynamical system simulation time units (t):

$$7.93 \cdot 10^{-6} \cdot \tau^{-1} \cdot t$$

The runtime multiplies the simulation time by the baseline integration speed of the HCDCv2 (7.93 μs) and time scale factor (`timescale statement`) of the ADP to compute the runtime in wall-clock seconds. The HCDCv2 has operates at a baseline frequency of `126000` Hz (`freq` statement). At this baseline speed, one unit of hardware integration time corresponds to $126000^{-1} Hz = 7.93\mu\ s$ of wall clock time.

The HCDCv2 runtime then invokes the `simulation execution` operation with the computed runtime in wall-clock seconds to run the provided ADP. This command powers on and observes the programmed circuit, then returns the measured waveform from the probed `cext` block at `(0,3,2,0)`.

The time and amplitude of the returned waveform are measured in wall-clock seconds and volts, respectively. The HCDCv2 runtime converts the time and the amplitude of the signal back into dynamical system units before returning the signal.

Time Recovery: The HCDCv2 runtime converts each time measurement t_i from wall clock time to dynamical system time with the following relation:

$$(7.93 \cdot 10^{-6} \cdot \tau^{-1})^{-1} \cdot t_i$$

The above relation multiplies each sample by the reciprocal of the time scale factor that maps dynamical system time to wall-clock time.

Amplitude Recovery: The HCDCv2 runtime identifies the scale factor associated with port `z` of the `cext` block at `(0,3,2,0)` in the ADP. The port scale factor is identified by an ADP statement of the form `scale z = a` in the `config` statement for block `(0,3,2,0)` of block `cext`. It divides each voltage measurement v_i by the port's scale factor `a` to recover the signal amplitude in dynamical system amplitude units.

5.10 Conclusion

Differential equation-solving analog devices are a promising new class of ultra-low-power computing platforms that solve dynamical systems energy efficiently. Internally, these devices contain programmable analog blocks and a programmable interconnect for forming circuits. This class of analog devices is digitally re-configurable and programmed by digitally routing signals between blocks and configuring individual blocks to implement the desired computations. In this thesis, I target the HCDCv2, a re-programmable differential-equation solving analog device that targets general non-linear dynamical systems.

The analog blocks present in these devices are subject to a range of low-level physical behaviors that affect the fidelity of the mapped computation. Relevant phenomena include analog noise, frequency- and input-dependent fixed errors, and unexpected gains and biases that are introduced into the block post-fabrication. Hardware designers eliminate unexpected gains and biases and some forms of static error through a process called calibration. The calibration procedure tunes the circuits in the device to eliminate unwanted behaviors. Traditionally, the calibration procedure aims to eliminate all unexpected behavioral deviations from each target block. In this work, I present an alternative co-designed calibration strategy that prioritizes eliminating unwanted behaviors that cannot be compensated for in compilation.

The compiler reasons about all other low-level block behaviors. The hardware designer provides frequency and operating range restrictions to the compiler to ensure the analog blocks are well-behaved and do not exhibit frequency- and value-dependent unwanted behaviors. The hardware designer also provides analog noise descriptions to the compiler. The compiler uses this information to produce circuits that respect the physical limitations of the system.

In practice, the calibration procedure may not fully eliminate the unexpected gains and biases present in the given block. In this chapter, I introduce the concept of a delta model. A delta model is a symbolic model which captures the unexpected behavior present in a block instance after calibration. The delta model parameters for the device at hand are derived by characterizing the calibrated blocks and fitting this characterization to a parameterized version of the delta model called the delta model specification. The compiler uses the delta models for the target device to more effectively target the device on hand.

In this chapter, I introduce an analog device specification language (ADSL) that enables the hardware designer to describe the programmable analog blocks and available digitally programmable interconnects to the compiler. The ADSL supports the specification of block programming interfaces, block input and output ports, and the block's input-output relations. The ADSL also supports the specification of operating range and frequency restrictions, analog noise, and quantization error. The ADSL offers language constructs for defining delta model specifications and delta model parameters.

I then introduce the analog device programming language (ADPL) that specifies analog circuits comprised of configured analog blocks. The compiler produces analog device programs written in the ADPL as compilation outputs. The ADPL offers language constructs for configuring analog blocks, routing signals between blocks, and specifying transformations that change the digital values written to the circuit.

The remainder of the chapter covers the operation of the HCDCv2. I first present the ADS for the HCDCv2. I then describe the calibration algorithms, runtime system, and low-level interface employed by the HCDCv2.

Chapter 6

Scaled and Unscaled ADPs

The compiler maps the target dynamical system computation to the analog hardware and generates, as output, a circuit composed of configured blocks. For the produced circuit to faithfully implement the target dynamical system computation, the circuit physics must preserve the original dynamical system dynamics such that the original dynamical system variable trajectories can be recovered at runtime from the voltage and current trajectories by applying a compiler-derived recovery transform. The compiler specifies the produced circuit and associated recovery transform in the analog device program language (ADPL) presented in Chapter 5.

Programming Challenges: Analog devices, including the HCDCv2, exploit the device physics to implement computation directly with physical signals. While this direct computation is the key to the energy efficiency of analog devices, it also requires produced ADP to operate successfully in the presence of a variety of challenging physical phenomena. The compiler must address these challenges when crafting a circuit:

- **Operating Ranges:** Physical properties such as voltage and currents have *operating ranges* that limit the range of values a signal or value may take on. Each voltage or current must fall within its respective operating range for the device to operate properly. If these operating ranges are violated, the component may become damaged or fail to perform the computation to specification. Each operating range is specified as a minimum and maximum signal value.

- **Manufacturing Variations:** Two blocks of the same type may not implement exactly the same set of functions in practice due to variations introduced in the manufacturing process. These manufacturing variations may cause the block's behavior to deviate from the promised specification. While hardware designers deploy hardware measures such as device calibration to mitigate the effects of process variation, these hardware measures do not always fully eliminate these variations in behavior.

- **Frequency Limitations:** Certain blocks may impose speed limitations on the target computation when operating under certain modes. The hardware designer imposes these frequency limitations to ensure the block doesn't operate in regimes where the frequency affects the block behavior.

- **Analog Noise and Digital Error**: All analog signals are subject to some form of analog noise. Similarly, all digital signals and fields are subject to quantization error. These sources of error may have an outsized effect on signals with small amplitudes and dynamic ranges. These noise and error characteristics may change depending on the block mode.

The compiler produces an unscaled ADP that directly maps the target dynamical system to the analog hardware a scaled ADP that respects the physical constraints of the analog hardware:

- **Unscaled ADP**: The unscaled ADP maps the dynamical system to the analog device without considering any of the physical restrictions and behaviors present in the device. The physics of the unscaled ADP exactly match the dynamics of the dynamical system, provided all of the blocks behave ideally under all conditions. The unscaled ADP typically cannot be directly run on the analog hardware since it does not consider the physical constraints of the device. The compiler produces the unscaled ADP during the circuit synthesis stage of compilation. Refer to Chapter 7 and Chapter 8 for more information on circuit synthesis.

- **Scaled ADP**: The scaled ADP scales all signals and values in the circuit so that it executes correctly given the operating range and frequency restrictions imposed on the hardware. The scaled ADP preserves the original dynamics of the dynamical system so that it can be recovered from the measured signals at runtime by applying an inverting transform. The scaled ADP also compensates for any behavioral deviations and attenuates the effects of noise and quantization error. The compiler produces each scaled ADP with a specific calibration strategy in mind – this is necessary because the behavioral deviations present in the calibrated blocks change depending on how the blocks are calibrated. Refer to Chapter 7 and Chapter 9 for more information on circuit synthesis.

  The scaled ADP defines a scaling transform and recovery transform for the circuit. The scaling transform comprises positive, constant magnitude scale factors that change the dynamic ranges and amplitudes of all the digital and analog signals and a time scale factor that changes the speed of the simulation. The magnitude scale factors are defined in the scaled ADP with `scale` statements and the time scale factor is defined in the scaled ADP with a `timescale` statement.

  The scaled ADP can be directly run on the analog hardware. The HCDCv2 runtime dispatches and executes the provided scaled ADP on the device on hand.

The compiler first produces an unscaled ADP which implements the target dynamical system. The unscaled ADP configures the data fields and mode of each block instance and routes the block instances together to form a circuit. The compiler then scales the unscaled ADP to produce the scaled ADP. While computing the scaling transform, the compiler partially reconfigures the blocks in the unscaled ADP to better scale the circuit. For example, the compiler may change the block mode or alter the expressions mapped to expression data fields to obtain a better scaling transform.

- **Cosine ADP**: I first walk through the unscaled and scaled ADPs for the cosine benchmark from Section 4.1. This example introduces all the concepts which will be used throughout the chapter.
- **Overview and Notation**: I present a comprehensive overview of how the unscaled and scaled ADPs are presented in this chapter. This overview also more formally introduces the notation used in the cosine program.
- **Unscaled ADPs**: I present the unscaled ADPs for each benchmark application from Sections 4.2-4.12. I discuss the complexity of the unscaled ADP and describe how the signals in the unscaled ADP relate to variables and expressions from the dynamical system. This analysis also examines the signal dynamics of the unscaled circuit. The signal dynamics are made up of symbolic expressions which describe the evolution of the currents and voltages in the ADP over time. I demonstrate that the signal dynamics are semantically equivalent to the dynamical system dynamics for the unscaled ADP.
- **Scaled ADPs for Dynamical System Applications**: I present the scaled ADPs for each benchmark application from Sections 4.2-4.12. I provide an overview of how the scaling transform scales the signals, values, and execution speed. For each scaled ADP, I present the scaled dynamics of the circuit and demonstrate that the scaled dynamic preserves the original dynamics of the system.
- **Discussion**: I discuss the overarching trends seen in the unscaled and scaled ADPs.

```
  var V = integ(-P,0.0);
  var P = integ(V, 1.0);
  var Position = emit(0.6*P);
  interval P = [-1,1];
  interval V = [-1,1];
  time 20;
```

Figure 6-1: Dynamical system specification for the `cos` benchmark

```
config block integ @ (0, 3, 2, 0) {
   modes [(m,m,+)];
   source V at z;
   set z0 at 0.000;
}
config block integ @ (0, 3, 1, 0) {
   modes [(m,m,+)];
   source P at z;
   set z0 at 0.500;
}
config block fanout @ (0, 3, 3, 1) {
   modes [(+,+,-,m), (+,+,-,h)];
   source P at x;
}
config block tout @ (0, 3, 0, 0) {
   modes [(*)];
}
config block extout @ (0, 3, 2, 0) {
   modes [(*)];
   source Position at z;
}
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 3, 1);
conn block fanout port z2 loc (0, 3, 3, 1) with block integ port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
```

Figure 6-2: Unscaled ADP for `cos` benchmark

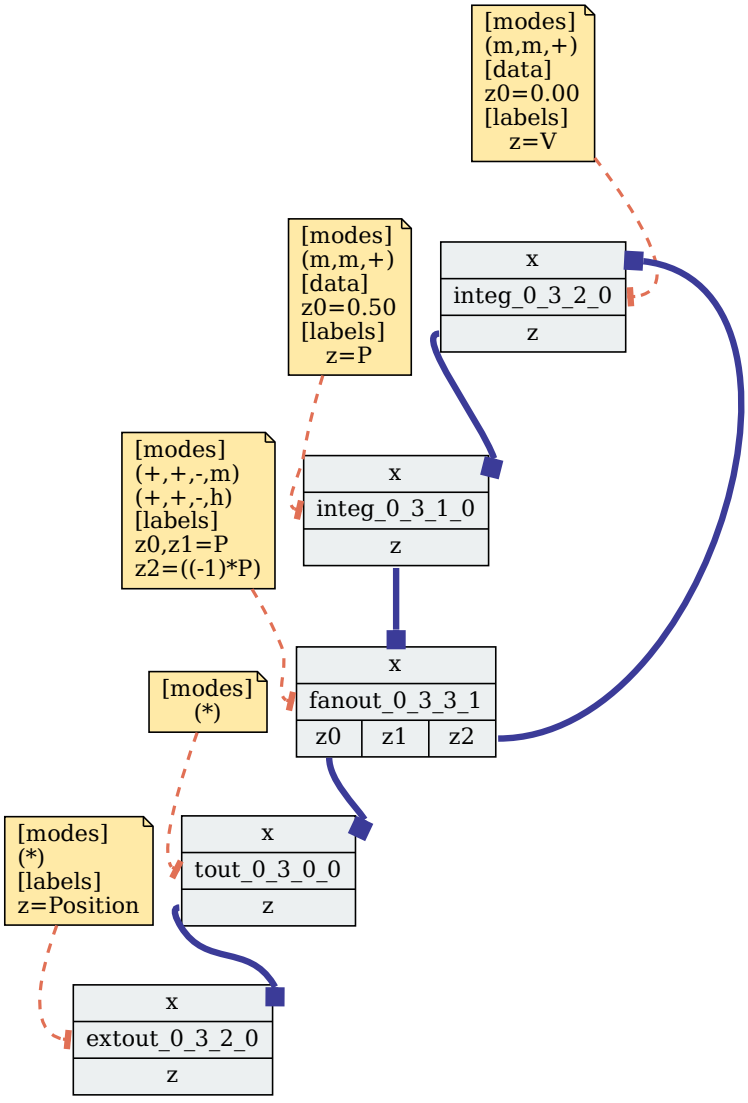


Figure 6-3: Circuit representation of unscaled ADP for the `cos` application

6.1 Simple Oscillator (cos)

Figure 6-1 presents the dynamical system for the simple oscillator application which was introduced in Section 4.1. The dynamical system defines the state variables `P` and `V` and the observation variable `Position` which observes the variable `P` over time.

Figure 6-2 presents the ADP which implements the simple oscillator dynamical system presented in Figure 6-1. Figure 6-3 presents a diagram of the circuit implemented by the unscaled ADP. The block instances (grey) have input ports (top boxes) and output ports (bottom boxes). The yellow callout boxes in the figure present the configuration written to each block in the ADP. The blue connections between ports route together the programmable blocks to implement the harmonic oscillator. Refer to Section 5.7 for a complete analog device specification for the HCDCv2 analog device. The above unscaled ADP performs the following operations:

- **Data Fields and Modes**: The compiler configures a subset of block instances available in the HCDCv2 to implement the desired functions. Each block instance configuration sets the modes and the data fields of the block. Lines 2,7,12,16, and 19 all set the modes for each block instance. Lines 4 and 9 set the constant data field values that determine the initial condition for the integrator blocks. The programming interface of each block is provided by the device ADS.
- **Connections**: The compiler connects together input and output ports by enabling digitally settable connections defined in the ADS. Lines 22-26 enable a subset of the available digitally programmable connections on the device.
- **Dynamical System Variables**: The compiler maps analog signals at ports in the ADP to dynamical system variables with `source` annotations. For each annotated port, the physics the signal at that port implements the dynamics of the mapped dynamical system variable or expression. Lines 8 and 3 map the the position `P` and velocity `V` to the analog currents produced by the integrator (`integ`) blocks at location `(0,3,1,0)` and `(0,3,2,0)` respectively. Line 13 maps the signal observation variable `Position` to the voltage at port `z` of the observation (`extout`) block.

The above ADP uses two integrator (`integ`) blocks and a signal observation (`extout`) block to implement the oscillator computation. The circuit also uses a current copier (`fanout`) block to copy and negate the position `P` – these positive and negative copies of `P` are used to implement the `Position` and `V` variables respectively. Note that analog currents must be copied to be used multiple times in a circuit. The unscaled ADP also introduces a `tout` route block to make the connection between the current copier and the observation block. This is necessary because the `fanout` block at `(0,3,3,1)` and `extout` block at `(0,3,2,0)` cannot be directly connected.

```
        V    = integ((-1*P),(2*0))
        P    = integ(V,(2*0.50))
 Position    = emit(0.60*P)
```

Figure 6-4: Unscaled dynamics of the `cos` benchmark

6.1.1 Signal Dynamics of Unscaled ADP

The dynamics of the unscaled ADP are guaranteed to implement the dynamics of the dynamical system presented in Figure 6-1. I derive a set of symbolic expressions which capture the time-varying dynamics of the signals moving through the ports which implement the `V`, `P`, and `Pos` dynamical system variables (Lines 3,8,20 of Figure 6-2). I derive each expression by traversing the unscaled ADP and propagating the mapped dynamical system variables through the block input-output relations through the circuit without simplification. The dynamics of the labeled signals are presented as a collection of variable-expression assignments.

Figure 6-4 presents the symbolic expressions governing the dynamics of the analog currents implementing `P`, `V`, and `Pos`. I derive each symbolic expression by propagating the dynamical system variables through the input-output relations implemented by each of the configured blocks. The above symbolic expressions contain digitally settable constant and expression data fields (`blue`), dynamical system variables and expressions (`dark blue`), and block dynamics (`black`)

The above signal expressions do not syntactically match the starting dynamical system expressions. The compiler instantiates the initial condition of the `P` variable to `0.50` to compensate for the device coefficient `2.0`. A *device coefficient* or device term is an algebraic term introduced by a block. The compiler cannot directly modify device coefficients. The above ADP also uses device coefficients when possible to implement dynamical system values. The compiler negates the `P` signal and implements the coefficient `0.60` in the `Position` signal expression by leveraging the existing block dynamics instead of introducing additional blocks.

Deriving the Dynamics of `V`

The ADP implements the variable `V` at output port `z` of the `integ` block at `(0,3,2,0)`. The current at `z` implements `integ(x,2*z0)` when the block mode is `(m,m,+)` and the data field `z0` is instantiated to `0.00`:

```
V = integ(x,2*0.000)
```

I next to derive the dynamics of the analog signal supplied to the input port `x`. Looking at the ADP, the negated position `(-1)*P` at port `z2` of the fanout block is routed to port `x` of the integrator block. The current at `z2` of the current copier implements `(-1)*x` when the block mode

is set to either `(+,+,-,m)` or `(+,+,-,h)` mode:

```
V = integ((-1)*P,2*0.000)
```

The above symbolic expression describes the dynamics of the signal implementing the velocity `V`. This symbolic expression perfectly captures the signal dynamics provided the HCDCv2 blocks perfectly implement the input-output relations from the ADS block specifications.

6.1.2 Challenges with Running the Unscaled ADP

In practice, the unscaled ADP does not faithfully implement the described dynamical system on an analog device for the following reasons:

- **Frequency Limit Violations**: The unscaled ADP directly maps the dynamical system simulation time to hardware time. With this mapping, one unit of simulation time corresponds to one unit of hardware integration time, which corresponds to 126000^{-1} units of wall-clock time. The integration speed of the unscaled circuit is, therefore, `126` kHz. This integration speed exceeds the maximum supported frequency of 80 kHz. The integrator blocks impose this frequency restriction in the circuit.
- **Not Recoverable**: The above circuit targets an idealized model of the HCDCv2 analog device. In this idealized model, each block of the same type in the circuit implements the same set of input-output relations. In practice, these blocks implement functions which deviate from the idealized dynamics of the block. For example, the integrators at `(0,3,2,0)` and `(0,3,1,0)` are both expected to implement `integ(x,2*z0)` when in `(m,m,+)` mode. However, in practice, these integrators implement `(integ(0.9821*x, 0.7978*2*z0)` and `integ(0.9786*x, 0.7417*2*z0)` respectively when in `(m,m,+)` mode. The unscaled circuit does not account for these behavioral variations which occur in practice in the device.

The unscaled ADP also fails to capitalize on the full abilities of the HCDCv2 platform. The signals implementing `V` and `P` have a dynamic range of [-1,1] μA and the signal implementing the `Position` variable has a dynamic range of [-0.60,0.60] V. The device supports analog currents ranging from [-2,2] μA and analog voltages ranging from [-1.2,1.2] V. The signals which implement variables in the unscaled ADP therefore only use half of the available dynamic range. This increases the impact of noise on the computation.

The compiler scales the input analog device program to mitigate the above issues. This procedure also modifies the mode for each block since the block mode affects the physical constraints and behaviors of that block.

```
config block integ @ (0, 3, 2, 0) {
   modes [(h,h,+)];
   scale x = 1.116; scale z = 3.624; scale z0 = 0.465;
   source V at z;
   set z0 at 0.000;
}
config block integ @ (0, 3, 1, 0) {
   modes [(h,m,+)];
   scale x = 3.624; scale z = 1.116; scale z0 = 1.885;
   source P at z;
   set z0 at 0.500;
}
config block fanout @ (0, 3, 3, 1) {
   modes [(+,+,-,m)];
   scale x,z0,z1 = 1.116;
   source P at x;
}
config block tout @ (0, 3, 0, 0) {
   modes [(*)];
   scale x,z = 1.116;
}
config block extout @ (0, 3, 2, 0) {
   modes [(*)];
   scale x,z = 1.116;
   source Position at z;
}
timescale 0.306674
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 3, 1);
conn block fanout port z2 loc (0, 3, 3, 1) with block integ port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
```

Figure 6-5: Scaled dynamics of the `cos` benchmark

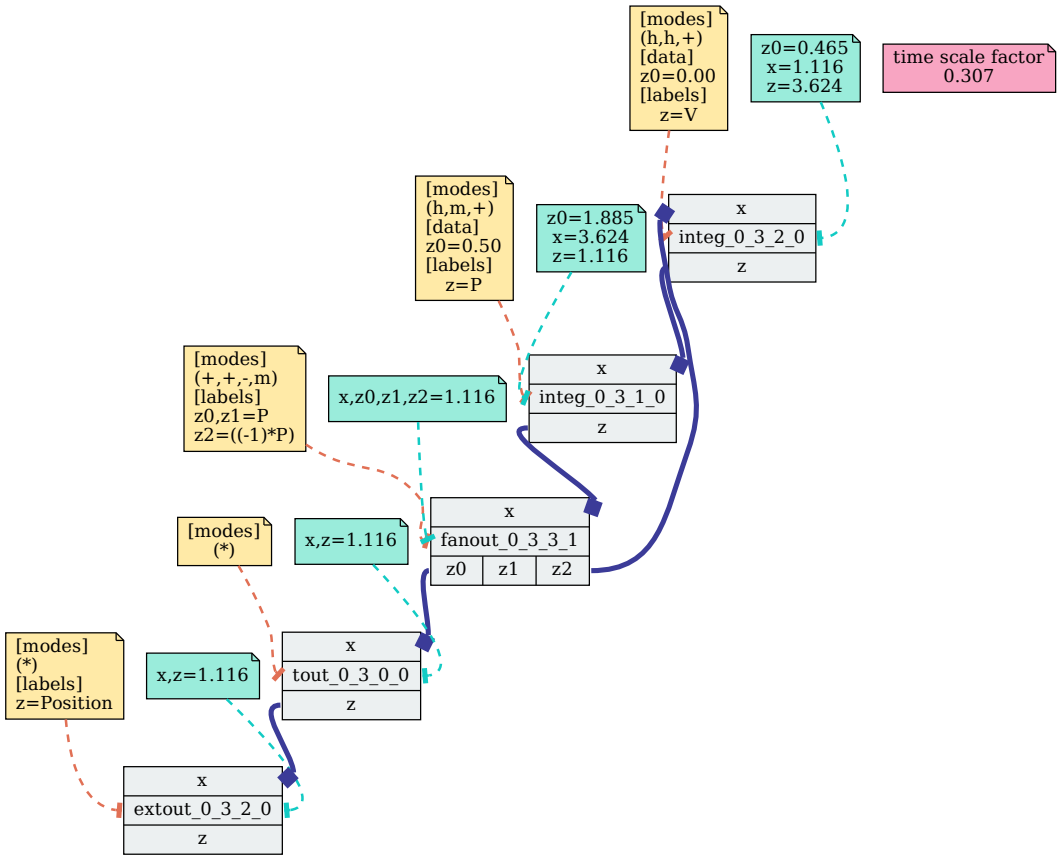


Figure 6-6: Circuit representation of scaled ADP for the `cos` benchmark

6.1.3 Scaled ADP

Figure 6-5 presents the scaled ADP for the harmonic oscillator. Figure 6-6 presents a diagram of the circuit implemented by the scaled ADP. The teal boxes present the magnitude scale factors for the port and data fields and the pink box presents the time scale factor. Lines 3,9,15,20, and 24 define the magnitude scale factors for the ports and data fields in the ADP. Line 27 defines the time scale factor for the ADP. The defined scale transform scales execution time by `0.306674` – the harmonic oscillator dynamics therefore evolve at 0.306674x the baseline execution speed of the system. The magnitude of the signals implementing the `V`, `P`, and `Position` variables are scaled by `3.624`, `1.116`, and `1.116` respectively. The compiler scales the values supplied to data field `z0` of integrators `(0,3,2,0)` and `(0,3,1,0)` by `0.465` and `1.885` respectively.

The scaled ADP changes the modes of the integrator blocks at `(0,3,2,0)` and `(0,3,1,0)` to from `(m,m,+)` to `(h,h,+)` and `(h,m,+)` respectively. The mode change from `(m,m,+)` to `(h,h,+)` scales the initial condition of the integrated signal by `10` and increases the operating range of the port `x` and `z` from [-2,2] to [-20,20]. The mode change from `(m,m,+)` to `(h,m,+)` scales the derivative of the signal by `0.1` and integrates the operating range of port `x` from [-2,2] to [-20,20]. The scaled ADP selects the `(+,+,-,m)` mode for the current copier block. This mode limits the operating range at ports `x`, `z0`, `z1`, and `z2` ports to [-2,2] μA. This compiler selects this mode from the `(+,+,-,m)` and `(+,+,,-,h)` modes specified in the unscaled ADP.

Execution: The HCDCv2 runtime executes the above scaled ADP by first applying the scaling transform and then configuring the device to implement the transformed ADP. The runtime then runs the configured circuit and records any externally accessible signals. For the above ADP, the runtime would record the signal implementing the `Position` variable from port `z` of the `extout(0,3,2,0)` block. The scaled ADP respects all of the physical constraints of the hardware during execution. After execution, runtime recovers the original dynamics of the variables from the recorded signals by inverting the scaling transform. I describe the scaling transform application and inversion operators below:

- **Applying the Transform**: The scaling transform is applied by multiplying each digitally settable data field by its magnitude scale factor. This automatically internally scales all the signals within the circuit by their associated magnitude scale factors and scales the simulation speed by the time scale factor. To apply the scaling transform in the above ADP, the data field value provided to port `z0` of integrator `integ(0,3,1,0)` is multiplied by 1.885 (`1.885*0.50 = 0.9425`) and the data field value provided to port `z0` of the integrator `integ(0,3,2,0)` is multiplied by 0.465 (`0*0.465=0.0`).

- **Inverting the Transform**: To recover the original dynamical system dynamics from the collected signals, the scaling transform must be inverted. To apply the recovery transform,

the compiler divides the signal by the associated port's magnitude scale factor and multiplies each time sample, measured in wall-clock time, by the ADP's time scale factor and the time constant defined in the ADS. This inverted scaling transform is also referred to as the recovery transform in this thesis. In the above ADP, the amplitude of the recorded `Position` signal is divided by `1.116` – the magnitude scale factor associated with port `z` of the `extout(0,3,2,0)` block. To recover simulation time, each time sample t_i is multiplied by the time scale factor `0.306674` and the hardware time constant `126000`. The conversion factor `0.306674*126000` translates the time samples from wall-clock time to dynamical system simulation time.

The scaled ADP has several key characteristics which enable it to account for the described physical behaviors while also preserving the original dynamical system dynamics. I summarize these properties below :

- **Physically Realizable** (Section 6.1.4): Each of the signals in the scaled ADP respects the operating range and frequency limitations of the port accepting the signal. The scaled circuit minimizes the effect of quantization error and analog noise on the computation.
- **Recoverable** (Section `6.1.5`): The original simulation can be recovered by scaling the magnitudes and times recorded at the sampled digital outputs by derived scaling factors. The scaled ADP preserves the original dynamics of the dynamical system. To ensure the original simulation is recoverable, the scaled ADP must preserve the original dynamics of the unscaled circuit. The scaled ADP compensates for any behavioral deviations present in the device on hand to preserve the original dynamics of the dynamical system. In the scaled ADP, the dynamics of each scaled signal can be written as the idealized unscaled signal dynamics of the signal times a constant coefficient. The idealized dynamics of each signal from the unscaled ADP can therefore be recovered by factoring out a constant coefficient from the scaled dynamics of the target signal.

  Note that because behavioral deviations change based on how the blocks are calibrated, the scale transform targets blocks calibrated with a specific calibration strategy. The `cos` application targets the `maximize_fit` calibration strategy provided by the HCDCv2.

These properties together enable the scaled ADP to be executed faithfully on the analog device and ensure the inverting transform successfully recovers the original dynamics.

6.1.4 Physical Realizability of Scaled ADP

The scaled ADP presented in Figure 6-5 respects all of the operating range limitations and frequency range limitations imposed by the HCDCv2:

- **Operating Range Limitations**: The magnitude of the signals implementing `V`, `P`, and `Position` are scaled by `3.624`, `1.116`, and `1.116` respectively. The dynamic range of the `V` signal changes from [-1,1] to [-3.624,3.624] and the dynamic range of the `P` signal changes from [-1,1] to [-1.116,1.116] with the above scale transform. The dynamic range of the `Position` signal changes from [-0.6,0.6] to [-0.6696,0.6696]. Because the signal implementing `V` is outside the operating range [-2,2], the compiler changes the block modes of the `integ` blocks at `(0,3,2,0)` and `(0,3,1,0)` to `(h,h,+)` and `(h,m,+)` respectively. These modes support signals between [-20,20] μA at ports `z` and `x` respectively. The scaled data field values `0.9425` and `0.00` both fall within the data field operating range of [-1,1].
- **Frequency Limitations**: The above circuit maps 3.26 units of hardware time to one unit of simulation time. This corresponds to a simulation speed of `0.306674*126` or `38.640` kHz. Therefore, the simulation speed of the scaled circuit does not exceed the maximum supported execution speed of 80 kHz. This frequency is the maximum frequency supported by the integrator block.

The scaled circuit scales the analog signals and digital values in the circuit to reduce the effect of quantization error and noise on the computation:

- **Maximize Data Field Values**: All of the non-zero, digitally settable multiplier coefficients are scaled to be as close to -1 or 1 as possible; these are the maximum supported digital values for the blocks. This scaling strategy reduces the effect of quantization error on the digital values. In the above scaled ADP, the data field value `0.50` (line 11) is scaled up from 0.50 to 0.9425.
- **Maximize Signal Dynamic Ranges**: All time-varying signals are scaled up to consume as much of the port operating ranges as possible. This scaling strategy improves the signal-to-noise ratio (SNR) for the scaled signals and reduces the effect of analog noise. The dynamic range of `V`, `P`, and `Position` signals are all scaled up so that they occupy a larger dynamic range.

```
        V_sc   = (3.6240*V)
               = integ((3.2608*(0.9954*(1.1165*((-1)*P)))),((7.8000*2)*(0.4646*0)))
        P_sc   = (1.1165*P)
               = integ((3.2608*(0.0945*(3.6240*V))),((0.5922*2)*(1.8852*0.5000)))
Position_sc    = (1.1165*Position) = (0.6000*emit((1.1165*P)))
```

Figure 6-7: Scaled dynamics of the `cos` benchmark

6.1.5 Preservation of Original Dynamics in Scaled ADP

The compiler identifies a scaling transform *preserves* the original dynamics of the dynamical system. The scale transform preserves the original dynamical system dynamics if the original dynamical system variable trajectories can be recovered at runtime by applying a statically derived inverting transform. For the derived scaling transform to preserve the original dynamical system dynamics, the scaled signal dynamics must simplify to the original unscaled signal dynamics times the magnitude scale factor of the variable. If this property holds, then the original dynamics can be recovered by applying the inverting transform. The computed scaling transform must also compensate for behaviors that may compromise the integrity of the scaled signals:

- **Behavioral Deviations in Calibrated Blocks**: The computed scaling transform must compensate for any empirically observed behavioral variations in the calibrated blocks. For the HCDCv2, these behavioral deviations for the target calibration strategy are retrieved from the delta model database introduced in Chapter 5. The scaling transform compensates for the correctable delta model parameters in each delta model.
- **Mode Changes**: The compiler occasionally changes the assigned block modes when scaling the ADP. In some cases, changing the block mode may also change the input-output relation implemented by the block. The computed scaling transform must correct for any changes in block behavior that arise from changing the block mode.

Figure 6-7 presents the scaled signal dynamics for the scaled signals $\texttt{V}_{sc}$, $\texttt{P}_{sc}$, and $\texttt{Positions}_{sc}$ (Lines 3,8,11 of Figure 6-5) which implement `V`, `P`, and `Position` dynamical system variables. Each of the above signal relations contains magnitude and time scale factors (`red`) and compensation terms (`grey`). The compensation terms capture the effect of empirically observed behavioral deviations and changes to the block mode on the computation.

- **Scaled Signals and Values**: The scaled signal relations reference two data field scale factors, three variable scale factors, and one time scale factor ($\texttt{3.624}^{-1}$ =`0.306`). The scaled circuit scales the data fields implementing `0` and `0.500` by `0.465` and `1.885` respectively. The scaled circuit scales the `V`, `P`, and `Position` variables by `3.624x`, `1.116x`, and `1.116x` respectively.
- **Compensation Terms**: The `0.995` and `0.592` compensation terms capture the empirically observed block instance-specific gain terms present in the integrator blocks. The `0.448e-02`

and 7.800 compensation terms capture both the effects of a block mode change and model the behavioral deviations found on the device. The 0.995 compensation term captures an imperfection in the calibrated block which causes the input signal x to be scaled 0.995. The 7.800 compensation term captures the effect of changing the block mode from (m,m,+) to (h,h,+) and the empirically observed behavior of the calibrated block. In the unscaled circuit, the initial condition of the integrated signal is 2.0*z0. In the scaled circuit presented above, the scaled initial condition is 20.0*0.78*z0; this is 7.8x larger than the initial condition of the unscaled circuit.

- **Execution Speed**: The speed of the scaled computation is 0.306x the baseline integration speed of the device. The scaled signals evolve 0.306x more slowly than the dynamical system dynamics, relative to the baseline integration speed of the device (as defined in the ADS). The compiler exploits a property of dynamical systems when designing the scaling transform – this property enables the compiler to tune the execution speed of the dynamical system. If all derivatives are scaled by some coefficient α, then the simulation evolves at αx the baseline execution speed. Refer to Section3.1.2 for more information on how the compiler is able to adjust the speed of the dynamical system.

  The 0.306^{-1}=3.2608 term sets the simulation speed by leveraging the above dynamical system property. This coefficient is introduced into the derivative of each state variable in the scaled ADS. The 3.6208 term sets the ratio between the scaled variable and the derivative of the scaled variable. In the above equations, the derivative of V is scaled by 0.9954*1.1165 = 1.111. This scale factor is 0.306x smaller than 3.624, the magnitude scale factor of V.

Signal Preservation: In the scaled ADP, each scaled signal preserves the original signal dynamics from the unscaled ADP (blue, dark blue, and black terms). For example, the original dynamics of V can be recovered from the scaled signal V_{sc} since the following equality relation holds:

V_{sc}=(3.6240*V)=integ((3.2608*(0.9954*(1.1165*((-1)*P)))),((7.8000*2)*(0.4646*0)))

I next show that this equality relation holds.I first factor out an expression of scale factors and compensation terms (red and grey terms) from the right-hand side of the relation.

V_{sc}=(3.6240*V) = integ((3.2608*0.9954*1.1165)*((-1)*P),((7.8000*0.4646)*(2*0)))

V_{sc}=(3.6240*V) = integ((3.6240)*((-1)*P),((3.6240)*(2*0)))

V_{sc}=(3.6240*V) = (3.6240)*integ(((-1)*P),2*0)

V_{sc}=(3.6240*V) = (3.6240)*V

I first factor out the scale factors and compensation terms from the derivative and initial condition expressions in the integration operator. The expression 3.2608*0.9954*1.1165 can be factored

out of the derivative of V_{sc}. This expression simplifies to 3.6240, the magnitude scale factor of V. The scale expression 7.800*0.4646 can be factored out of the initial value V(0) of V_{sc}. This expression simplifies to 3.624, the magnitude scale factor of V.

I then simplify these expressions to the constant value 3.624. Because both the initial condition and derivative are scaled by the same amount, I can factor out the 3.624 from the integration operator. The integ((-1)*P, 2*0) expression matches the signal dynamics for V presented in Figure 6-4. I am therefore able to manually confirm that the right-hand side of the scaled signal relation equals the unscaled dynamics of the signal times an expression of symbolic terms which simplifies to the magnitude scale factor 3.624. The scaled signal dynamics presented above, therefore, preserve the original dynamics of the velocity of the harmonic oscillator.

Preservation of Signal Dynamics in Scaled ADP

The compiler produces scaled ADPs that preserve the original signal dynamics from the unscaled ADP. This preservation property also preserves the dynamical system dynamics. For the cos application, the compiler produces a scaled ADP which preserves the original dynamical system dynamics presented in Figure 6-2. I can confirm that the scaled ADP preserves the original dynamics by analyzing the scaled signals presented in Figure 6-5:

- V **variable**: The scale expressions for the derivative and initial condition of the signal both simplify to 3.624, the magnitude scale factor for V:

$$V_{sc}\text{'} = 3.6240*V\text{'} \quad = 3.261*0.995*1.116*((-1)*P)$$
$$V_{sc}(0) = 3.6240*V(0) \quad = 7.800*0.465*0.0$$

- P **variable**: The scale factors for the derivative and initial condition of the signal both simplify to 1.116, the magnitude scale factor of P:

$$P_{sc}\text{'} = 1.116*P\text{'} \quad = 3.261*9.448e\text{-}02*3.624*V$$
$$P_{sc}(0) = 1.116*P(0) \quad = 0.592*1.885*0.5$$

- Position **variable**: The scale expression factored from the dynamics of the signal matches the magnitude scale factor of the Position variable:

$$\text{Position}_{sc} = 1.116*\text{Position} \quad = 0.6*(1.116*P)$$

6.2 Notation and Overview

The remainder of this chapter presents the unscaled and scaled signal dynamics for each of the benchmarks for each benchmark application presented in Chapter 4. I present the following for each

benchmark application:

- **Original Dynamical System**: I present the original dynamical system dynamics. Each dynamical system is written in the dynamical system specification language introduced in Chapter 3.
- **Unscaled and Scaled ADP**: I present the unscaled and scaled ADPs for the target applications. I discuss the complexity of the unscaled ADP and describe any mode changes made in the scaled ADP.
- **Unscaled and Scaled Signal Dynamics**: I present the unscaled and scaled signal dynamics for the target applications. I discuss the magnitude and time scale factors for each scaling transform and discuss what behaviors each of the compensation terms captures.
- **Demonstration of Signal Preservation**: I demonstrate that the computed scaling transform preserves the dynamics of the unscaled circuit.

6.2.1 Unscaled Signal Dynamics

This analysis presents the dynamics of each signal in the unscaled ADP. The unscaled signal dynamics contains the following kinds of terms:

- **Data Field Values** (`blue`): The signal dynamics contains the values written to digitally-programmable constant and expression data fields.
- **Dynamical System Variables**(`dark blue`): The signal dynamics contains dynamical system variables and expressions. These variables represent analog signals at ports in the ADP with `source` annotations.
- **Block Dynamics**(`black`): The signal dynamics contains block dynamics, as defined by the input-output relations in the ADS. Coefficients and terms which are part of the block dynamics are referred to as device coefficients or device terms in this analysis. The compiler must therefore compensate for these device coefficients. The produced unscaled ADPs compensate for these behaviors in a few ways:

6.2.2 Syntactic Matching

I often discuss to what extent the signal dynamics syntactically match the dynamical system dynamics. For a signal expression to syntactically match a dynamical system expression, it must use the exact same operators, variables, and values in exactly the same order. Term reordering is only allowed for associative operators. For example, `A+B` and `B+A` syntactically match for this definition of syntactic matching.

6.2.3 Scaled Signal Dynamics

This analysis also presents the dynamics of each signal in the scaled ADP. Like the unscaled signal dynamics, the scaled signal dynamics contain data field values, dynamical system variables, and terms describing the block dynamics. The scaled dynamics of each signal also introduces the following new constructs:

- **Time and Magnitude Scale Factors** (red): The compiler derives a scaling transform comprised of time and magnitude scale factors. The scale transform is applied by multiplying each data field by a magnitude scale factor. All other scale factors which appear in the scaled signal expression are implicitly set by applying the magnitude scale factors to the ADP data fields.
- **Injected Coefficients**(purple): the compiler injects coefficients into the expressions implemented by expression data fields to more freely scale the ADP. See the `pend` application for an example of how scaling transform uses injected coefficients.
- **Compensation Terms** (grey): The scaled signal dynamics incorporate compensation terms that model the effect of empirically observed behavioral variations and changes to the block mode on the block's behavior.

The scaled signal expression matches the unscaled signal expression if all of the compensation terms, scale factors, and injected variables are eliminated or factored out of the expression.

6.2.4 Signal Preservation

The original dynamical system dynamics can be recovered from the scaled adp at runtime because the scaled ADP preserves the original dynamical system dynamics. The compiler produces scaled ADPs which preserve the signal dynamics from the unscaled ADP and the original dynamical system dynamics. The preservation property is described below:

- **Preservation Property**: Consider a relation describing the scaled signal dynamics of the form $\texttt{V}_{sc}$ `=` `x*V` `=` $\texttt{E}_{sc}$ where `V` is a dynamical system variable and `x` is a positive, real number. The scaled signal dynamics perserves the original dynamics of the variable `V` `=` `E` if a constant coefficient `y` can be factored out of `E` such that `y*E=` $\texttt{E}_{sc}$ and `y=x`.

For each application, I show how the scaled signal dynamics preserves the original behavior of the unscaled circuit. Because of the way I choose to formulate the scaled circuit dynamics in this chapter, I can show that the `y*E=`$\texttt{E}_{sc}$ property holds by factoring out an expression $\texttt{E}_y$ which contains all of the scale factors, compensation terms, and injected coefficients from the scaled signal expression $\texttt{E}_{sc}$. The $\texttt{E}_y$ expression simplifies to the constant value `y`. I present a shorthand for

demonstrating signal preservation for four types of signals (I-IV). This shorthand is used to more compactly show preservation for the benchmark applications:

(type I) Variable Assignment: Consider a scaled variable-expression assignment with no addition operators V_{sc} = x*V = E_sc which preserves the original dynamics of the signal V = E. For each signal of this form, I factor out an expression E_y of scale factors, compensation terms, and injected coefficients from E_{sc} such that E_y*E = E_{sc}. I can then simplify E_y to obtain the constant value scf(y) which equals scf(x). Provided E_y = x and E_y*E = E_{sc}, this scaled signal expression preserves the original dynamics of the signal:

1 V_{sc} = x*V = E_{sc}

2 V_{sc} = x*V = E_y*E

3 V_{sc} = x*V = y*E

4 V_{sc} = x*V = x*E

5 V_{sc} = x*V = x*V

I compactly represent the above derivation for signals of this type with the following notation:

V x = E_y

(type II) Variable Assignment with Addition: Consider a scaled variable-expression assignment with addition operators V_{sc} = x*V = $E_{sc,1}$+$E_{sc,2}$+...+$E_{sc,n}$ which preserves the original dynamics of the signal V = E_1+E_2+...+E_n. For each $E_{sc,i}$ expression, I factor out an expression $E_{y,i}$ of scale factors, compensation terms, and injected coefficients such that $E_{y,i}$*E_i = $E_{sc,i}$. I can then simplify each $E_{y,i}$ to obtain the constant value y which equals x. Provided $E_{y,i}$ = x and $E_{y,i}$*E_i = $E_{sc,i}$, than the sum of signals preserves the original dynamics:

1 V_{sc} = x*V = $E_{sc,1}$+$E_{sc,2}$+...+$E_{sc,n}$

2 V_{sc} = x*V = $E_{y,1}$*E_1+$E_{y,2}$*E_2+...+$E_{y,n}$*E_n

3 V_{sc} = x*V = y_1*E_1+y_2*E_2+...+y_n*E_n

4 V_{sc} = x*V = x*E_1+x*E_2+...+x*E_n

5 V_{sc} = x*V = x*(E_1+E_2+...+E_n)

6 V_{sc} = x*V = x*V

I compactly represent the above derivation for signals of this type with the following notation:

E_1 x = $E_{y,1}$

...

E_n x = $y_{y,n}$

(type III) Differential Equation:Consider a scaled differential equation V_{sc} = x*V = integ($E_{sc,1}$,$E_{sc,2}$) which preserves the original dynamics of the signal V = integ(E_1,E_2). For each signal $E_{sc,i}$, I factor out an expression E_y,i of scale factors, compensation terms, and injected coefficients such

that $E_{y,i}$*E_i = $E_{sc,i}$. I can then simplify each $E_{y,i}$ to obtain the constant value y which equals x. Provided $E_{y,1}$ = $E_{y,2}$ = x, $E_{y,2}$*E_2 = $E_{sc,2}$, and $E_{y,1}$*E_1 = $E_{sc,1}$, than the integrated signal preserves the original dynamics:

V_{sc} = x*V = integ($E_{sc,1}$,$E_{sc,2}$)

V_{sc} = x*V = $\int E_{sc,1}$ $V_{sc}(0)$ = x*V(0) = $E_{sc,2}$

V_{sc} = x*V = $\int E_{y,1}$*E_1 $V_{sc}(0)$ = x*V(0) = $E_{y,2}$*E_2

V_{sc} = x*V = $\int y_1$*E_1 $V_{sc}(0)$ = x*V(0) = y_2*E_2

V_{sc} = x*V = x*($\int E_1$) $V_{sc}(0)$ = x*V(0) = x*(E_2)

V_{sc} = x*V x*integ(E_1,E_2)

V_{sc} = x*V x*V

I compactly represent the above derivation for signals of this type with the following notation:

V x = $E_{y,1}$

V(0) x = $E_{y,2}$

(type IV) Differential Equation with Addition:Consider a scaled differential equation V_{sc} = x*V = integ($E_{sc,1}$+...+$E_{sc,n}$,E_{n+1}) which preserves the original dynamics of the signal V = integ(E_1+...+E_n,E_{n+1}). For each scaled expression $E_{sc,i}$, I factor out an expression $E_{y,i}$ of scale factors, compensation terms, and injected coefficients for both the derivative expressions such that $E_{y,i}$*E_i = $E_{sc,i}$. Each expression $E_{y,i}$ simplifies to the constant value y_i which equals x. Provided $E_{y,i}$ = x and $E_{y,i}$*E_i = $E_{sc,i}$, then the integrated signal preserves the original dynamics:

V_{sc} = x*V = $\int E_{sc,1}$+...+$E_{sc,n}$ $V_{sc}(0)$ = x*V(0) = $E_{sc,n+1}$

V_{sc} = x*V = $\int E_{y,1}$*E_1+...+$E_{y,n}$*E_n $V_{sc}(0)$ = x*V(0) = $E_{y,n+1}$*E_{n+1}

V_{sc} = x*V = $\int y_1$*E_1+...+y_n*E_n $V_{sc}(0)$ = x*V(0) = y_{n+1}*E_{n+1}

V_{sc} = x*V = $\int$ x*E_1+...+x*E_n $V_{sc}(0)$ = x*V(0) = x*E_{n+1}

V_{sc} = x*V = $\int$ x*(E_1+...+E_n) $V_{sc}(0)$ = x*V(0) = x*E_{n+1}

V_{sc} = x*V = x*integ(E_1+...+E_n,E_{n+1})

I compactly represent the above derivation for signals of this type with the following notation:

E_1 x = $E_{y,1}$

...

E_n x = $E_{y,n}$

V(0) x = $E_{y,n+1}$

Note that the signal preservation rules are guaranteed to hold if the involved expression E equals zero. This is because zero remains the same regardless of what it is multiplied with.

```
var V = integ(-0.22*V - 0.84*P,-2.0);
var P = integ(V, 9.0);
var Position = emit(P);
interval P = [-10,10];
interval V = [-10,15];
time 20;
```

Figure 6-8: Dynamical system specification for `cosc` benchmark

```
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source V at z; set z0 at -1.000; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source P at z; set z0 at 4.500; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.840; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -0.220; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source Position at z; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source V at z0; source V at z1;
  source V at z2; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source P at z0; source P at z1;
  source P at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 1) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 2, 1) with block integ port x loc (0, 3, 0, 0);
```

Figure 6-9: Unscaled ADP of `cosc` benchmark

```
V = integ((((-0.84)*P))+(((-0.22)*V)),(2*(-1)))
P = integ(V,(2*4.50))
Position = emit((0.60*(1.67*P)))
```

Figure 6-10: Signal dynamics of unscaled ADP for `cosc` benchmark

```
config block integ @ (0, 3, 2, 0) {
  modes [(h,m,+)]; scale x = 0.811; source V at z; scale z = 0.121; set z0 at -1.000;
  scale z0 = 0.135; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; scale x = 0.121; source P at z; scale z = 0.188; set z0 at 4.500;
  scale z0 = 0.209; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,h)]; scale x = 0.188; scale y = 1.000; scale z = 0.811; set c at -0.840;
  scale c = 0.439; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,h)]; scale x = 0.121; scale y = 1.000; scale z = 0.811; set c at -0.220;
  scale c = 0.714; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m)]; scale x = 0.188; scale y = 1.000; scale z = 0.105; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.105; source Position at z; scale z = 0.105; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,m)]; scale x = 0.121; source V at z0; scale z0 = 0.121;
  source V at z1; scale z1 = 0.121; source V at z2; scale z2 = 0.121; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,+,+,m)]; scale x = 0.188; source P at z0; scale z0 = 0.188;
  source P at z1; scale z1 = 0.188; source P at z2; scale z2 = 0.188; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.105; scale z = 0.105; }
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 1) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 3, 1);
conn block fanout port z0 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z1 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 2, 1) with block integ port x loc (0, 3, 0, 0);
timescale 0.634921
```

Figure 6-11: Scaled ADP of `cosc` benchmark

$$V_{sc} = (0.1206*V) = integ((1.5750*(0.0944*((9.8309*((0.4392*(-0.8400))*(0.1878*P))))+((9.4169*((0.7139*(-0.2200))*(0.1206*V))))))),((0.8906*2)*(0.1354*(-1))))$$

$$P_{sc} = (0.1878*P) = integ((1.5750*(0.9885*(0.1206*V))),((0.8964*2)*(0.2095*4.5000)))$$

$$Position_{sc} = (0.1052*Position) = (0.6000*emit((0.9906*((0.5656*1.6665)*(0.1878*P)))))$$

Figure 6-12: Scaled dynamics for the `cosc` benchmark

6.3 Dynamical System Applications

I next present the unscaled and scaled ADPs for each benchmark application from Sections 4.2-4.12. For each unscaled ADP, I discuss the complexity of the circuit and describe how the signals in the unscaled ADP relate to variables and expressions from the dynamical system. For each scaled ADP, For each scaled ADP, I discuss the characteristics of the scaling transform, present the scaled dynamics of the circuit, and demonstrate that the scaled signal dynamics preserves the original dynamics of the system.

6.3.1 Dampened Harmonic Oscillator (cosc)

Figure 6-8 presents the dynamical system specification for the cosc application. The cosc application defines the P and V state variables and the Position variable:

```
var V = integ(-0.22*V - 0.84*P,-2.0);
var P = integ(V, 9.0);
var Position = emit(P);
```

The Unscaled ADP

Figure 6-9 presents the unscaled ADP for the cosc application. The ADP uses three multipliers, two current copiers, two integrators, one routing (tout) block, and one observation (extout) block. Lines 21-30 enable the nine connections necessary to form the desired circuit. The circuit contains has five constant data fields which provide the values -1, 4.5, -0.840, -0.220, and 1.667 to the circuit.

Figure 6-10 presents the physics of the labeled signals at lines 2,4,11 of the unscaled ADP. The ADP implements V and P with analog currents and Position with an analog voltage:

```
V = integ((((-0.84)*P)+((-0.22)*V)),(2*(-1)))
P = integ(V,(2*4.50))
Position = emit((0.60*(1.67*P)))
```

The dynamics of the V, P, and Position signals all fail to syntactically match the dynamical system dynamics presented above.

- V: The compiler implements the initial condition 2 as 2*-1, where 2 is a device term and -1 is a data field. This signal is the analog current at port z of integrator (0,3,2,0).
- P: The compiler implements the initial condition 9 as 2*4.5, where 2 is a device term and 4.5 is a data field. This signal is the analog current at port z of integrator (0,3,0,0).

- Position(observation block (0,3,2,0),port z): The compiler introduces the 1.67 data field to compensate for the 0.60 device term. This signal is the analog voltage at port z of observation block (0,3,2,0).

The Scaled ADP

Figure 6-11 presents the scaled ADP for the cosc application. The scaled ADP contains 5 data field scale factors, three variable scale factors, and one time scale factor. The scaled ADP defines a total of 30 magnitude scale factors and one time scale factor (0.6349). The speed of the scaled computation is therefore 0.6349x the baseline integration speed of the device. The compiler also selectively changes the block modes to more effectively scale the circuit. Some of these block mode modifications change the input-output relation implemented by the block. The block mode changes which change the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 2, 0)	[(m,m,+)]	(h,m,+)
mult	(0, 3, 3, 0)	[(x,m,m), (x,h,h)]	(x,m,h)
mult	(0, 3, 2, 1)	[(x,m,m), (x,h,h)]	(x,m,h)

Columns 1 and 2 specify the block instances, and Columns 3 and 4 present the modes assigned to the block instance in the unscaled and scaled ADPs, respectively. The scaling transform compensates for all changes to the input-output relation. Each of these mode changes alters the input-output relation implemented at the output ports of the block:

- **integrator** (0,3,2,0): The compiler changes the mode from (m,m,+) to (h,m,+). This modification scales the derivative signal by 0.1 and increases the supported operating range of the derivative signal.
- **multipliers**: The compiler changes the multiplier modes from (x,m,m) or (x,h,h) to (x,m,h). This modification scales the output signal by 10 but reduces the supported operating range of the input signal (relative to (x,h,h)).

Figure 6-12 presents the physics of the scaled signals implementing the V, P, and Position dynamical system variables:

```
V_sc = (0.1206*V) = integ((1.5750*(0.0944*((9.8309*((0.4392*(-0.8400))*(0.1878*P)))
        +(9.4169*((0.7139*(-0.2200))*(0.1206*V))))))),((0.8906*2)*(0.1354*(-1))))
P_sc = (0.1878*P) = integ((1.5750*(0.9885*(0.1206*V))),((0.8964*2)*(0.2095*4.5000)))
Position_sc = (0.1052*Position) = (0.6000*emit((0.9906*((0.5656*1.6665)*(0.1878*P)))))
```

The compiler scales the V, P, and Position signals by 0.1206,0.1878, and 0.1052 respectively. The compiler scales the data fields values 0.84, -0.22, -1, 4.5, and 1.667 by 0.439, 0.714, 0.135, 0.209, and 0.566 respectively.

All compensation terms capture the behavioral deviations present in the device. The 0.0945 compensation term additionally captures the effect of the integrator mode change on the computation. The 9.831 and 9.417 compensation terms additionally capture the effect of changing the multiplier mode on the computation. The 0.891, 0.989, and 0.896 compensation terms only capture the behavioral deviations of the blocks.

Preservation: The scale factors (red) and compensation terms (grey) can be eliminated from the physics of each scaled signal:

- V **variable** (**IV**): The scale factor for each term in the derivative and the initial condition all simplify to 0.121, the magnitude scale factor of the variable V:

$$\begin{aligned} -0.84*P \quad 0.121 &= 1.575*9.445e\text{-}2*9.831*0.439*0.188 \\ -0.22*V \quad 0.121 &= 1.575*9.445e\text{-}2*9.417*0.714*0.121 \\ V(0) \quad 0.121 &= 0.891*0.135 \end{aligned}$$

- P **variable** (**III**): The scale factor for the derivative and the the initial condition all simplify to 0.188, the magnitude scale factor of the variable P:

$$\begin{aligned} P' \quad 0.188 &= 1.575*0.989*0.121 \\ P(0) \quad 0.188 &= 0.896*0.209 \end{aligned}$$

- Position **variable** (**I**): The signal expression for the Position signal simplifies to 0.1052, the magnitude scale factor for the position variable Position:

$$\text{Position} \quad 0.1052 = 0.9906*0.5656*0.1878$$

```
func sinf(T) = sin(T)
var angvel = integ(-0.18*angvel-0.8*call(sinf,ang),-1.0);
var ang = integ(angvel, 1.0);
var Angle = emit(ang);
interval ang = [-1.5,1.5];
interval v = [-1.5,1.5];
time 20;
```

Figure 6-13: Dynamical system specification for pend benchmark

```
config block lut @ (0, 3, 2, 0) {
  modes [(*)]; set e at (1*(0.05*((-0.80)*sin((2*(1*y))))));  }
config block adc @ (0, 3, 2, 0) { modes [(m)]; }
config block dac @ (0, 3, 2, 0) {
  modes [(dyn,h)]; set c at 0.000; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source angvel at z; set z0 at -0.500; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source ang at z; set z0 at 0.500; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.180; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.000; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source Angle at z; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source angvel at z0; source angvel at z1;
  source angvel at z2; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source ang at z0; source ang at z1;
  source ang at z2; }
config block tout @ (0, 3, 0, 0) { modes [(*)]; }
conn block adc port z loc (0, 3, 2, 0) with block lut port x loc (0, 3, 2, 0);
conn block lut port z loc (0, 3, 2, 0) with block dac port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 3, 0);
conn block dac port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 3, 1);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 3, 1) with block adc port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 3, 1);
```

Figure 6-14: Unscaled ADP of pend benchmark

```
angvel = integ((((-0.18)*angvel))+((20*(0.05*((-0.80)*sin((2*(0.50*ang)))))))),
         (2*(-0.50)))
ang = integ((1*angvel),(2*0.50))
Angle = emit((0.60*(1.67*ang)))
```

Figure 6-15: Signal dynamics of unscaled ADP for for pend benchmark

```
config block lut @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 1.257; scale z = 23.564;
  set e at (23.5645*(0.0500*((-0.8000)*sin((2*(0.7957*y))))))); scale e = 23.564; }
config block adc @ (0, 3, 2, 0) {
  modes [(m)]; scale x = 1.204; scale z = 1.257; }
config block dac @ (0, 3, 2, 0) {
  modes [(dyn,m)]; scale x = 23.564; scale z = 2.176; set c at 0.000; scale c = 1.000; }
config block integ @ (0, 3, 3, 0) {
  modes [(h,m,+)]; scale x = 2.176; source angvel at z; scale z = 0.653;
  set z0 at -0.500; scale z0 = 0.712; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; scale x = 0.390; source ang at z; scale z = 1.204; set z0 at 0.500;
  scale z0 = 1.299; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m)]; scale x = 0.653; scale y = 1.000; scale z = 2.176; set c at -0.180;
  scale c = 3.353; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m)]; scale x = 0.653; scale y = 1.000; scale z = 0.390; set c at 1.000;
  scale c = 0.601; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m)]; scale x = 1.204; scale y = 1.000; scale z = 0.679; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.679; source Angle at z; scale z = 0.679; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m)]; scale x = 0.653; source angvel at z0; scale z0 = 0.653;
  source angvel at z1; scale z1 = 0.653; source angvel at z2; scale z2 = 0.653; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,+,+,m)]; scale x = 1.204; source ang at z0; scale z0 = 1.204;
  source ang at z1; scale z1 = 1.204; source ang at z2; scale z2 = 1.204; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.679; scale z = 0.679; }
conn block adc port z loc (0, 3, 2, 0) with block lut port x loc (0, 3, 2, 0);
conn block lut port z loc (0, 3, 2, 0) with block dac port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 3, 0);
conn block dac port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 3, 1);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 3, 1) with block adc port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 3, 1);
timescale 0.317460
```

Figure 6-16: Scaled ADP of pend benchmark

$$\text{angvel}_{sc} = (0.6530*\text{angvel}) = \text{integ}((3.1500*(0.0953*((0.9939*((3.3529*(-0.1800))*(0.6530*\text{angvel})))+((0.0923*20)*(23.5645*(0.0500*((-0.8000)*\sin((2*(0.7957*((1.0441*0.5000)*(1.2036*\text{ang}))))))))))),((0.9173*2)*(0.7119*(-0.5000))))$$

$$\text{ang}_{sc} = (1.2036*\text{ang}) = \text{integ}((3.1500*(0.9791*(0.9939*((0.6013*0.9999)*(0.6530*\text{angvel}))))),((0.9264*2)*(1.2992*0.5000)))$$

$$\text{Angle}_{sc} = (0.6790*\text{Angle}) = (0.6000*\text{emit}((0.9975*((0.5656*1.6665)*(1.2036*\text{ang})))))$$

Figure 6-17: Scaled dynamics for the pend benchmark

6.3.2 Pendulum (pend)

Figure 6-13 presents the dynamical system specification for the pend application. The pend application defines the angvel and ang state variables and the Angle variable. The dynamical system uses the sinf(T) function which computes the sin function (sin(T)) of an input:

```
func sinf(T) = sin(T)
var angvel = integ(-0.18*angvel-0.8*call(sinf,ang),-1.0);
var ang = integ(angvel, 1.0);
var Angle = emit(ang);
```

The Unscaled ADP

Figure 6-14 presents the unscaled ADP for the pend application. The ADP uses three multipliers, two current copiers, two integrators, one routing (tout) block, one observation (extout) block, one analog-to-digital converter (ADC, adc), one digital-to-analog converter (DAC, dac), and one lookup table lut). The compiler uses the route block to forward the signal implementing ang to the observation block. The current copiers are used to produce two copies of P and V respectively. The compiler implements the addition operator from the relation governing V with Kirchhoff's law. To add the signals together compiler forwards all of the signals implementing the addition terms to the same port. The signal flowing into the block from that port implements the sum of all the addition terms.

Lines 15-27 enable the twelve connections necessary to form the desired circuit. The circuit contains has six constant data fields which provide the values -0.50, 0.50,-0.180, 1.00, and 1.667.

The lut block implements the function e(x) where e is a programmable expression data field and x is a time-varying digital input. The compiler configures the data field e to implement the expression 0.05*(-0.80)*sin(2*x). The compiler integrates the lut block into the rest of the circuit using an analog-to-digital converter and digital-to-analog converter.

Figure 6-15 presents the physics of the unscaled ADP for the pend application. The ADP implements the ang, angvel variables with analog currents and the Angle variable with an analog voltage:

```
1 angvel = integ((((-0.18)*angvel)
2          +(20*(0.05*((-0.80)*sin((2*(0.50*ang)))))),(2*(-0.50)))
3 ang = integ((1.00*angvel),(2*0.50))
4 Angle = emit((0.60*(1.67*ang)))
```

The dynamics of ang, angvel and Angle all fail to syntactically match the dynamical system dynamcis presented above:

- `angvel`: The compiler inserts the `0.05` and `2.0` constants into the expression assigned to the expression data field. The compiler includes these values to compensate for the device coefficients introduced by the `adc` and `dac` blocks. The compiler also sets the initial condition of the integrator to `0.5` instead of `-1` to account for the `2.0` device coefficient introduced by the integrator. This signal is the analog current at port `z` of integrator `(0,3,3,0)`.
- `ang`: The compiler sets the initial condition of the integrator to `0.5` instead of `1.0` to account for the `2.0` device coefficient introduced by the integrator. The compiler also introduces a `1.0` coefficient into the derivative of the `ang` variable. The compiler later modifies this coefficient to more freely scale the ADP. This signal is the analog current at port `z` of integrator `(0,3,1,0)`.
- `Angle`: This signal is the analog voltage at port `z` of observation block `(0,3,2,0)`. The compiler introduces the `1.67` data field value to to account for the `0.60` device coefficient introduced by the observation block.

Scaled ADP

Figure 6-16 presents the scaled ADP for the `pend` application. The scaled ADP defines a total of 39 magnitude scale factors. The scaled ADP contains 5 data field scale factors, two injected coefficients, and one time scale factor (`0.3174`). The injected coefficients are introduced into the expression mapped to the expression data field to scale the data field argument `x` and the computed result. The speed of the computation is 0.3174x the baseline integration speed of the device. I summarize all of the mode changes which change the input-output relations of the blocks below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
dac	(0, 3, 2, 0)	[(dyn,h)]	(dyn,m)
integ	(0, 3, 3, 0)	[(m,m,+)]	(h,m,+)

All changes to the block input-output relations are compensated for by the scaling transform. Each of these mode changes alters the input-output relations implemented at the output ports of the blocks:

- **DAC** `(0,3,2,0)`: The compiler changes the mode from `(dyn,h)` to `(dyn,m)`. This modification scales the output signal by `0.1` and reduces the operating range of the output sginal.
- **integrator** `(0,3,3,0)`: The compiler changes the mode from `(m,m,+)` to `(h,m,+)`. This modification scales the derivative of the output signal by `0.1` and reduces the operating range of the output signal.

Figure 6-17 presents the physucs of the scaled signals implementing the `angvel`, `ang`, and `Angle` dynamical system variables:

```
1  angvel_sc = (0.6530*angvel) = integ((3.1500*(0.0953*((0.9939*((3.3529*(-0.1800))*(0.6530*angvel)))
2          +((0.0923*20)*(23.5645*(0.0500*((-0.8000)*sin((2*(0.7957*
3          ((1.0441*0.5000)*(1.2036*ang))))))))))),((0.9173*2)*(0.7119*(-0.5000))))
4  ang_sc = (1.2036*ang) = integ((3.1500*(0.9791*(0.9939*((0.6013*0.9999)*(0.6530*angvel))))),
5          ((0.9264*2)*(1.2992*0.5000)))
6  Angle_sc = (0.6790*Angle) = (0.6000*emit((0.9975*((0.5656*1.6665)*(1.2036*ang)))))
```

The compiler scales the magnitude of the angvel, ang, and Angle variables by 0.653, 1.204, and 0.679 respectively. The compiler scales the data fields -0.18, -0.50, 1.0,0.50, and 1.667 by 3.353, 0.712, 0.601, 1.299, and 0.566 respectively. It scales the speed of the computation by 3.150^{-1} or 0.317.

The compiler modifies the expression assigned to the expression data field from 0.05*(-0.80) sin((2*x)) to 23.5645*0.05*(-0.80)*sin((2*(0.7957*x). The compiler injects the 0.7957 and 23.5645 values into the expression to scale the argument x and the output of the expression respectively. The 0.7957 injected value cancels out the effect of both the scaling transform and the compensation terms from the scale expression. The 0.7957*1.0441*1.2036 term simplifies to 1.0, eliminating the scale transform and behavioral variations from the input argument. Because the the input signal is effectively unscaled, the compiler doesn't need to propagate the scaling transform through the expression data field dynamics. The compiler scales the data field result by 23.5646x by inserting the value 23.5645 into the expression.

The compensation terms with values between 0.917-1.044 only capture the behavioral variations in the device. The 0.0953 and 0.0923 compensation terms capture the effect of the mode changes described above. All other compensation terms capture both the effects of changing the block mode and model the behavioral deviations found on the device.

Preservation: The scale factors (red), injected coefficients (purple), and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- angvel **variable** (**IV**): The terms in the derivative and the scale factor in the initial condition all simplify to 0.653, the angvel magnitude scale factor :

```
-0.18*(angvel)   0.653   = 3.150*9.526e-02*0.994*3.353*0.653
 0.80*sin(ang)   0.653   = 3.150*9.526e-2*9.235e-02*23.564
   angvel(0)     0.653   = 0.917*0.712
```

  The compiler does not need to propagate the scale transform and compensation terms through the data field expression because it cancels out the effects of the scaling transform and compensation terms on each expression data field input. The scaled signal implementing angvel supplies the expression data field input x with the scaled signal 1.0441*0.5000*1.2036*ang.

The compiler multiplies the data field input `x` with `0.7957` to cancel out the effects of the scaling transform and compensation terms on the input:

```
1.0   = 0.7957*1.0441*1.2036
```

The product of the compensation terms, injected values, and scale factors factored out of the expression data field argument `x` simplifies to 1. The input `x` provided to the data field expression is, therefore, an unscaled input. The injected `23.564` term scales the value computed by the expression data field by 23.564.

- `ang` **variable** (**III**): The scale factor for the derivative and the initial condition both simplify to `1.204`, the scale factor for the `ang` variable:

```
-0.18*(ang)   1.204   = 3.150*0.979*0.994*0.601*0.653
   ang(0)     1.204   = 0.926*1.299
```

- `Angle` **variable** (**I**): The scale expression for the signal simplifies to `0.679`, the scale factor of the `Angle` variable:

```
Angle   0.679   = 0.997*0.566*1.204
```

```
func frc(T) = sgn(T)*sqrt(abs(T))
var fPA = call(frc,PA)
var fPB = call(frc,PB)
var VA = integ(0.5*fPB - fPA - 0.15*VA,0.0);
var VB = integ(0.5*fPA - fPB - 0.15*VB,0.0);
var PA = integ(VA,2.0);
var PB = integ(VB,-1.0);
var PosA = emit(PA);
interval PA = [-2.5,2.5];
interval VA = [-2.5,2.5];
interval PB = [-2.5,2.5];
interval VB = [-2.5,2.5];
time 20;
```

Figure 6-18: Dynamical system specification for `spring` benchmark.

```
conn block adc port z loc (0, 0, 2, 0) with block lut port x loc (0, 0, 2, 0);
conn block lut port z loc (0, 0, 2, 0) with block dac port x loc (0, 0, 2, 0);
conn block adc port z loc (0, 0, 0, 0) with block lut port x loc (0, 0, 0, 0);
conn block lut port z loc (0, 0, 0, 0) with block dac port x loc (0, 0, 0, 0);
conn block mult port z loc (0, 0, 3, 0) with block integ port x loc (0, 0, 3, 0);
conn block mult port z loc (0, 0, 1, 1) with block integ port x loc (0, 0, 3, 0);
conn block mult port z loc (0, 0, 0, 1) with block integ port x loc (0, 0, 3, 0);
conn block mult port z loc (0, 0, 0, 0) with block integ port x loc (0, 0, 1, 0);
conn block mult port z loc (0, 0, 1, 0) with block integ port x loc (0, 0, 1, 0);
conn block mult port z loc (0, 0, 2, 1) with block integ port x loc (0, 0, 1, 0);
conn block mult port z loc (0, 3, 0, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block dac port z loc (0, 0, 2, 0) with block fanout port x loc (0, 0, 2, 0);
conn block dac port z loc (0, 0, 0, 0) with block fanout port x loc (0, 0, 3, 0);
conn block integ port z loc (0, 0, 3, 0) with block fanout port x loc (0, 0, 3, 1);
conn block integ port z loc (0, 0, 1, 0) with block fanout port x loc (0, 0, 0, 0);
conn block integ port z loc (0, 0, 0, 0) with block fanout port x loc (0, 0, 2, 1);
conn block fanout port z0 loc (0, 0, 2, 1) with block adc port x loc (0, 0, 2, 0);
conn block fanout port z1 loc (0, 0, 2, 1) with block tout port x loc (0, 0, 0, 0);
conn block tout port z loc (0, 0, 0, 0) with block tin port x loc (0, 3, 0, 0);
conn block tin port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 0, 2, 0) with block adc port x loc (0, 0, 0, 0);
conn block fanout port z0 loc (0, 0, 3, 0) with block mult port x loc (0, 0, 3, 0);
conn block fanout port z1 loc (0, 0, 3, 0) with block mult port x loc (0, 0, 0, 0);
conn block fanout port z0 loc (0, 0, 3, 1) with block mult port x loc (0, 0, 1, 1);
conn block fanout port z1 loc (0, 0, 3, 1) with block integ port x loc (0, 0, 0, 0);
conn block fanout port z0 loc (0, 0, 2, 0) with block mult port x loc (0, 0, 0, 1);
conn block fanout port z1 loc (0, 0, 2, 0) with block mult port x loc (0, 0, 2, 1);
conn block fanout port z0 loc (0, 0, 0, 0) with block mult port x loc (0, 0, 1, 0);
conn block fanout port z1 loc (0, 0, 0, 0) with block integ port x loc (0, 0, 2, 0);
```

Figure 6-19: Connections for unscaled and scaled ADPs of `spring` benchmark

```
config block lut @ (0, 0, 2, 0) {
  modes [(*)]; set e at (1*(0.05*(pow(|(2*(1*y))|,0.50)*sgn((2*(1*y))))))); }
config block lut @ (0, 0, 0, 0) {
  modes [(*)]; set e at (1*(0.05*(pow(|(2*(1*y))|,0.50)*sgn((2*(1*y))))))); }
config block adc @ (0, 0, 2, 0) { modes [(m)]; }
config block adc @ (0, 0, 0, 0) { modes [(m)]; }
config block dac @ (0, 0, 2, 0) { modes [(dyn,h)]; source fPA at z; set c at 0.000; }
config block dac @ (0, 0, 0, 0) { modes [(dyn,h)]; source fPB at z; set c at 0.000; }
config block integ @ (0, 0, 3, 0) { modes [(m,m,+)]; source VA at z; set z0 at 0.000; }
config block integ @ (0, 0, 1, 0) { modes [(m,m,+)]; source VB at z; set z0 at 0.000; }
config block integ @ (0, 0, 0, 0) { modes [(m,m,+)]; source PA at z; set z0 at 1.000; }
config block integ @ (0, 0, 2, 0) { modes [(m,m,+)]; source PB at z; set z0 at -0.500; }
config block mult @ (0, 0, 3, 0) { modes [(x,m,m), (x,h,h)]; set c at 0.500; }
config block mult @ (0, 0, 1, 1) { modes [(x,m,m), (x,h,h)]; set c at -0.150; }
config block mult @ (0, 0, 0, 1) { modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 0, 0, 0) { modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 0, 1, 0) { modes [(x,m,m), (x,h,h)]; set c at -0.150; }
config block mult @ (0, 0, 2, 1) { modes [(x,m,m), (x,h,h)]; set c at 0.500; }
config block mult @ (0, 3, 0, 0) { modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) { modes [(*)]; source PosA at z; }
config block fanout @ (0, 0, 2, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source fPA at z0; source fPA at z1;
  source fPA at z2; }
config block fanout @ (0, 0, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source fPB at z0; source fPB at z1;
  source fPB at z2; }
config block fanout @ (0, 0, 3, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source VA at z0; source VA at z1;
  source VA at z2; }
config block fanout @ (0, 0, 0, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source VB at z0; source VB at z1;
  source VB at z2; }
config block fanout @ (0, 0, 2, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source PA at z0; source PA at z1;
  source PA at z2; }
config block tout @ (0, 3, 0, 0) { modes [(*)]; }
config block tout @ (0, 0, 0, 0) { modes [(*)]; }
config block tin @ (0, 3, 0, 0) { modes [(*)]; }
```

Figure 6-20: Block configurations for unscaled ADP of `spring` benchmark. See Figure 6-19 for connection statements.

```
fPA = (20*(0.05*(pow(|(2*(0.50*PA))|,0.50)*sgn((2*(0.50*PA))))))
fPB = (20*(0.05*(pow(|(2*(0.50*PB))|,0.50)*sgn((2*(0.50*PB))))))
VA = integ(((0.50*fPB)+(((-0.15)*VA) +((-1.00)*fPA))),(2*0))
VB = integ((((-1.00)*fPB)+(((-0.15)*VB) +(0.50*fPA))),(2*0))
PA = integ(VA,(2*1))
PB = integ(VB,(2*(-0.50)))
PosA = emit((0.60*(1.67*PA)))
```

Figure 6-21: Signal Dynamics for the `spring` unscaled ADP

```
config block lut @ (0, 0, 2, 0) {
  modes [(*)]; scale x = 0.805; scale z = 12.710;
  set e at (12.7097*(0.0500*(pow(|(2*(1.2425*y))|,0.5000)*sgn((2*(1.2425*y)))))); scale e = 12.710; }
config block lut @ (0, 0, 0, 0) {
  modes [(*)]; scale x = 0.591; scale z = 15.392;
  set e at (15.3922*(0.0500*(pow(|(2*(1.6908*y))|,0.5000)*sgn((2*(1.6908*y)))))); scale e = 15.392; }
config block adc @ (0, 0, 2, 0) {
  modes [(h)]; scale x = 7.557; scale z = 0.805; }
config block adc @ (0, 0, 0, 0) {
  modes [(m)]; scale x = 0.560; scale z = 0.591; }
config block dac @ (0, 0, 2, 0) {
  modes [(dyn,h)]; scale x = 12.710; source fPA at z; scale z = 11.659; set c at 0.000;
  scale c = 1.000; }
config block dac @ (0, 0, 0, 0) {
  modes [(dyn,h)]; scale x = 15.392; source fPB at z; scale z = 14.201; set c at 0.000;
  scale c = 1.000; }
config block integ @ (0, 0, 3, 0) {
  modes [(h,m,+)]; scale x = 1.705; source VA at z; scale z = 1.113; set z0 at 0.000;
  scale z0 = 1.333; }
config block integ @ (0, 0, 1, 0) {
  modes [(h,m,+)]; scale x = 1.344; source VB at z; scale z = 0.843; set z0 at 0.000;
  scale z0 = 1.077; }
config block integ @ (0, 0, 0, 0) {
  modes [(h,h,+)]; scale x = 1.113; source PA at z; scale z = 7.557; set z0 at 1.000;
  scale z0 = 0.943; }
config block integ @ (0, 0, 2, 0) {
  modes [(h,m,+)]; scale x = 0.843; source PB at z; scale z = 0.560; set z0 at -0.500;
  scale z0 = 0.751; }
config block mult @ (0, 0, 3, 0) {
  modes [(x,h,m)]; scale x = 14.201; scale y = 1.000; scale z = 1.705; set c at 0.500;
  scale c = 1.222; }
config block mult @ (0, 0, 1, 1) {
  modes [(x,m,m)]; scale x = 1.113; scale y = 1.000; scale z = 1.705; set c at -0.150;
  scale c = 1.594; }
config block mult @ (0, 0, 0, 1) {
  modes [(x,h,h)]; scale x = 11.659; scale y = 1.000; scale z = 1.705; set c at -1.000;
  scale c = 0.147; }
config block mult @ (0, 0, 0, 0) {
  modes [(x,h,m)]; scale x = 14.201; scale y = 1.000; scale z = 1.344; set c at -1.000;
  scale c = 0.950; }
config block mult @ (0, 0, 1, 0) {
  modes [(x,m,m)]; scale x = 0.843; scale y = 1.000; scale z = 1.344; set c at -0.150;
  scale c = 1.630; }
config block mult @ (0, 0, 2, 1) {
  modes [(x,h,m)]; scale x = 11.659; scale y = 1.000; scale z = 1.344; set c at 0.500;
  scale c = 1.812; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,h,m)]; scale x = 7.557; scale y = 1.000; scale z = 0.410; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.410; source PosA at z; scale z = 0.410; }
config block fanout @ (0, 0, 2, 0) {
  modes [(+,+,+,h)]; scale x = 11.659; source fPA at z0; scale z0 = 11.659;
  source fPA at z1; scale z1 = 11.659; source fPA at z2; scale z2 = 11.659; }
config block fanout @ (0, 0, 3, 0) {
  modes [(+,+,+,h)]; scale x = 14.201; source fPB at z0; scale z0 = 14.201;
  source fPB at z1; scale z1 = 14.201; source fPB at z2; scale z2 = 14.201; }
config block fanout @ (0, 0, 3, 1) {
  modes [(+,+,+,m)]; scale x = 1.113; source VA at z0; scale z0 = 1.113;
  source VA at z1; scale z1 = 1.113; source VA at z2; scale z2 = 1.113; }
config block fanout @ (0, 0, 0, 0) {
  modes [(+,+,+,m)]; scale x = 0.843; source VB at z0; scale z0 = 0.843;
  source VB at z1; scale z1 = 0.843; source VB at z2; scale z2 = 0.843; }
config block fanout @ (0, 0, 2, 1) {
  modes [(+,+,+,h)]; scale x = 7.557; source PA at z0; scale z0 = 7.557;
  source PA at z1; scale z1 = 7.557; source PA at z2; scale z2 = 7.557; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.410; scale z = 0.410; }
config block tout @ (0, 0, 0, 0) {
  modes [(*)]; scale x = 7.557; scale z = 7.557; }
config block tin @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 7.557; scale z = 7.557; }
timescale 0.148403
```

Figure 6-22: Block configurations and `timescale` statement for scaled ADP of `spring` benchmark. See Figure 6-19 for connection statements.

```
1  fPA_sc = (11.6594*fPA) = ((0.9174*20)*(12.7097*(0.0500*(pow(|(2*(1.2425*
2           ((0.1065*0.5000)*(7.5575*PA)))|,0.5000)*sgn((2*(1.2425*((0.1065*0.5000)*(7.5575*PA))))))))))
3  fPB_sc = (14.2009*fPB) = ((0.9226*20)*(15.3922*(0.0500*(pow(|(2*(1.6908*
4           ((1.0567*0.5000)*(0.5597*PB)))|,0.5000)*sgn((2*(1.6908*((1.0567*0.5000)*(0.5597*PB))))))))))
5  VA_sc = (1.1131*VA) = integ((6.7384*(0.0969*((0.0983*((1.2222*0.5000)*(14.2009*fPB)))
6           +((0.9608*((1.5945*(-0.1500))*(1.1131*VA)))
7           +(0.9929*((0.1473*(-1.0000))*(11.6594*fPA)))))))),((0.8349*2)*(1.3332*0)))
8  VB_sc = (0.8431*VB) = integ((6.7384*(0.0931*((0.0997*((0.9500*(-1.0000))*(14.2009*fPB)))
9           +((0.9785*((1.6298*(-0.1500))*(0.8431*VB)))
10          +(0.0636*((1.8123*0.5000)*(11.6594*fPA)))))))),((0.7827*2)*(1.0772*0)))
11 PA_sc = (7.5575*PA) = integ((6.7384*(1.0076*(1.1131*VA))),((8.0179*2)*0.9426))
12 PB_sc = (0.5597*PB) = integ((6.7384*(0.0985*(0.8431*VB))),((0.7449*2)*(0.7514*(-0.5000))))
13 PosA_sc = (0.4100*PosA) = (0.6000*emit((0.0959*((0.5656*1.6667)*(7.5575*PA)))))
```

Figure 6-23: Scaled signal dynamics for `spring` benchmark

6.3.3 Spring (spring)

Figure 6-18 presents the original dynamical system for the spring application. The spring application defines VA, VB, PA, and PB state variables and PosA, fPA, and fPB intermediate variables:

```
1    func frc(T) = sgn(T)*sqrt(abs(T))
2    var fPA = call(frc,PA)
3    var fPB = call(frc,PB)
4    var VA = integ(0.5*fPB - fPA - 0.15*VA,0.0);
5    var VB = integ(0.5*fPA - fPB - 0.15*VB,0.0);
6    var PA = integ(VA,2.0);
7    var PB = integ(VB,-1.0);
8    var PosA = emit(PA);
```

The spring application uses the frc function which computes the signed square root $sgn(T) \cdot \sqrt{|\,T\,|}$. The frc function is invoked with the PA and PB state variables as arguments.

Unscaled ADP

Figure 6-20 and Figure 6-20 present the unscaled ADP for the spring application. The ADP has a total of 26 blocks and 30 connections. The circuit contains 7 multipliers, 4 integrators, 2 ADCs, 2 DACs, 2 LUTs, 5 current copiers, 1 observation block, and 3 routing (tin and tout) blocks. the compiler uses the route blocks to forward the signal to the HCDCv2 tile (0,3) so that it can be provided to the extout block. The compiler uses the five copiers to produce two copies of the signals implementing fPA, fPB, PA, VA, VB. The compiler uses Kirchhoff's law to implement the addition operators seen in VA and VB with digitally settable connections. The ADP contains 11 constant data fields and 2 expression data fields. The integrator data fields provide the values 0.0, 1.0, and -0.50 to the circuit. The multiplier data fields provide the values 0.5, -0.15, -1.0, -1.0, -0.15, 0.5, and 1.667 to the circuit. The ADP contains 2 expression data fields which are both configured to implement the expression 0.05*pow(|(2*x)|,0.5)*sgn(2*x)), where x is the input value provided to the expression data field.

Figure 6-21 presents the physics of the labelled signals from the unscaled ADP. The ADP implements the fPA and fPB as time-varying digital signals, VA, VB, PA, and PB as analog currents, and PosA as an analog voltage:

```
1  fPA = (20*(0.05*(pow(|(2*(0.50*PA))|,0.50)*sgn((2*(0.50*PA))))))
2  fPB = (20*(0.05*(pow(|(2*(0.50*PB))|,0.50)*sgn((2*(0.50*PB))))))
3  VA = integ(((0.50*fPB)+(((-0.15)*VA) +((-1.00)*fPA))),(2*0))
4  VB = integ((((-1.00)*fPB)+(((-0.15)*VB) +(0.50*fPA))),(2*0))
```

```
PA = integ(VA,(2*1))
PB = integ(VB,(2*(-0.50)))
PosA = emit((0.60*(1.67*PA)))
```

The fPA, fPB, VA, VB, PA, PB, and PosA signal expressions syntactically do not match the original dynamical system relations:

- fPA and fPB: The compiler introduces the 2 and 0.5 values into the data field expression to account for the 0.50 and 20 device terms introduced by the ADC and DAC blocks respectively. The fPA and fPB signals are the time-varying digital signals at port z of DAC (0,0,2,0) and port z of DAC (0,0,0,0).
- VA: The compiler implements the subtraction operation with a combination of addition (via Kirchhoff's law) and signal negation. The fPA and 0.15*VA terms are negated and summed together instead of subtracted from the 0.50*fPB term. The initial condition does not need to be adjusted because 2*0 is 0. This signal is the analog current at port z of integrator (0,0,3,0).
- VB: The compiler implements the subtraction operation with a combination of addition (via Kirchhoff's law) and signal negation. The fPB and 0.15*VB terms are negated and summed together instead of subtracted from the 0.50*fPA term. The initial condition does not need to be adjusted because 2*0 is 0. This signal is the analog current at port z of integrator (0,0,1,0).
- PA: The compiler implements the initial condition 2 as 2*1 where 2 is a device term and 1 is a data field. This signal is the analog current at port z of integrator (0,0,0,0).
- PB: The compiler implements the initial condition -1 as 2*-0.5 where 2 is a device term and -0.5 is a data field. This signal is the analog current at port z of integrator (0,0,2,0).
- PosA: The compiler introduces the 1.67 data field to compensate for the 0.60 device term. This signal is the analog voltage at port z of observation block (0,3,2,0).

Scaled ADP

Figure 6-19 and Figure 6-22 present the scaled ADP for the spring application. The scaled ADP defines a total of 39 magnitude scale factors. The scaled ADP specifies eleven data field scale factors, six injected coefficients, seven variable scale factors, and one time scale factor (0.148). The speed of the computation is 0.148x the baseline integration speed of the device. The compiler also changes the block modes for a subset of the ADP blocks. The block mode changes which alter the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
adc	(0, 0, 2, 0)	[(m)]	(h)
integ	(0, 0, 3, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 0, 1, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 0, 0, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 0, 2, 0)	[(m,m,+)]	(h,m,+)
mult	(0, 0, 3, 0)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 0, 0, 0)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 0, 2, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 0, 0)	[(x,m,m), (x,h,h)]	(x,h,m)

- **multipliers**: The compiler changes the mode from `(x,m,m)` or `(x,h,h)` to `(x,h,m)`. This modification enables the compiler to multiply the input signal by a smaller constant value. This mode change scales the output signal by `0.1` and reduces the operating range of the output port (relative to the `(x,h,h)` mode).
- **ADC** `(0,0,2,0)`: The compiler changes the mode from `(m)` to `(h)`. The modification scales the output signal by `0.1` and increases the operating range of the input port.
- **integrators** `(0,0,3,0)`, `(0,0,1,0)`, **and** `(0,0,2,0)`: The compiler changes the mode from `(m,m,+)` to `(h,m,+)`. This modification scales the derivative of the output signal by `0.1` and increases the operating range of the block input port.
- **integrator** `(0,0,0,0)`: The compiler changes the mode from `(m,m,+)` to `(h,h,+)`. This modification scales the initial value of the signal by `10` and increases the operating range of both the block input and output ports.

Each of these mode changes alters the input-output relation implemented at the output ports of the block. The scaling transform compensates for these changes to the input-output relation.

Figure 6-23 presents the dynamics of the scaled signals from the scaled ADP for the `spring` benchmark:

```
fPA_sc = (11.6594*fPA) = ((0.9174*20)*(12.7097*(0.0500*(pow(|(2*(1.2425*
         ((0.1065*0.5000)*(7.5575*PA)))),0.5000)*sgn((2*(1.2425*((0.1065*0.5000)*(7.5575*PA))))))))))
fPB_sc = (14.2009*fPB) = ((0.9226*20)*(15.3922*(0.0500*(pow(|(2*(1.6908*
         ((1.0567*0.5000)*(0.5597*PB)))),0.5000)*sgn((2*(1.6908*((1.0567*0.5000)*(0.5597*PB))))))))))
VA_sc = (1.1131*VA) = integ((6.7384*(0.0969*((0.0983*((1.2222*0.5000)*(14.2009*fPB)))
         +((0.9608*((1.5945*(-0.1500))*(1.1131*VA)))
         +(0.9929*((0.1473*(-1.0000))*(11.6594*fPA)))))))),((0.8349*2)*(1.3332*0)))
VB_sc = (0.8431*VB) = integ((6.7384*(0.0931*((0.0997*((0.9500*(-1.0000))*(14.2009*fPB)))
         +((0.9785*((1.6298*(-0.1500))*(0.8431*VB)))
         +(0.0636*((1.8123*0.5000)*(11.6594*fPA)))))))),((0.7827*2)*(1.0772*0)))
PA_sc = (7.5575*PA) = integ((6.7384*(1.0076*(1.1131*VA))),((8.0179*2)*0.9426))
PB_sc = (0.5597*PB) = integ((6.7384*(0.0985*(0.8431*VB))),((0.7449*2)*(0.7514*(-0.5000))))
PosA_sc = (0.4100*PosA) = (0.6000*emit((0.0959*((0.5656*1.6667)*(7.5575*PA)))))
```

The compiler scales the magnitude of the fPA, fPB, VA, VB, PA, PB, and PosA variables by 11.659, 14.201, and 1.113, 0.843, 7.577, 0.560, and 0.410 respectively. The compiler reports a time scale factor of 0.148. The compiler therefore changes the speed of the computation by a factor of 6.738^{-1} or 0.148. The scaled signal dynamics use eleven constant data field scale factors, six injected coefficients, seven variable scale factors, and one time scale factor 0.148. I summarize the scaled data field values below:

- In the fPA signal, the compiler injects coefficients 1.2425 and 12.7097 into the expression data field. The 1.2425 coefficient eliminates the scaling transform and compensation terms from the input argument. The 12.7097 coefficient scales the data field expression result by a factor of 12.7097x.
- In the fPB signal, the compiler injects coefficients 1.6908 and 15.3922 into the expression data field. The 1.6908 coefficient eliminates the scaling transform and compensation terms from the input argument. The 15.3922 coefficient scales the data field expression result by a factor of 15.3922x.
- In the VA signal, the 0.50, -0.15, -1.0, and 0 data field values are scaled by 1.222, 1.594, 0.147, and 1.333 respectively.
- In the VB signal, -1.0, -0.15, 0.5, and 0 data fields are scaled by 0.950, 14.201, 1.630, 1.812, and 1.077 respectively.
- In the PA and PB signals, the 1 and -0.5 data field values are scaled by 0.943 and 0.751 respectively.
- The 1.667 data field value in the PosA signal is scaled by 0.566.

The compensation terms with values between 0.745-1.057 capture only behavioral variations in the device. All other compensation terms capture both the effect of changing the block mode and model the behavioral deviations found on the device. The compensation terms between 0.0636-0.1065 capture the mode changes for modes which introduce the 0.1 coefficient into the input-output relation. The compensation term 8.0197 captures the mode changes for the mode which introduces 10.0 coefficient into the input-output relation.

Preservation: The scale factors (red), injected variables (purple), and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- fPA **variable** (**I**): The signal expression for the signal dynamics simplifies to 11.659, the scale factor for the fPA variable.

$$\texttt{fPA} \quad 11.6594 \quad = 0.917 * 12.710$$

The compiler does not need to propagate the scale transform and compensation terms through the data field expression because it cancels out the effects of the scaling transform and compensation terms on each expression data field input. The signal implementing `fPA` supplies the expression data field input `x` with the scaled signal `0.1065*0.5000*7.5575*PA`. The compiler multiplies the data field input `x` with `1.2425` to cancel out the effects of the scaling transform and compensation terms on the input:

```
1.0   = 1.2425*0.1065*7.5575
```

The product of the compensation terms, injected values, and scale factors factored out of the expression data field argument `x` simplifies to 1. The input `x` provided to the data field expression is, therefore, an unscaled input. The injected `23.564` term scales the value computed by the expression data field by 23.564x.

- `fPB` **variable** (**I**): The signal expression for the signal dynamics simplifies to `14.201`, the scale factor for the variable `fPB`:

```
fPB   14.2009   = 0.923*15.392
```

The compiler does not need to propagate the scale transform and compensation terms through the data field expression because it cancels out the effects of the scaling transform and compensation terms on each expression data field input. The signal implementing `fPB` supplies the expression data field input `x` with the scaled signal `1.0567*0.5000*0.5597*PB`. The compiler multiplies the data field input `x` with `1.6908` to cancel out the effects of the scaling transform and compensation terms on the input:

```
1.0   = 1.6908*1.0567*0.5597
```

The product of the compensation terms, injected values, and scale factors factored out of the expression data field argument `x` simplifies to 1. The input `x` provided to the data field expression is, therefore, an unscaled input. The injected `15.392` term scales the value computed by the expression data field by 15.392x.

- `VA` **variable** (**IV**): The scale expressions for the terms in the derivative and the initial condition all simplify to `1.113`, the magnitude scale factor of the `VA` variable:

```
 0.5*fPB   1.1131   = 6.738*9.687e-02*9.825e-02*1.222*14.201
-0.15*VA   1.1131   = 6.738*9.687e-02*0.961*1.594*1.113
-1.0*fPA   1.1131   = 6.738*9.687e-02*6.363e-02*1.812*11.659
```

- `VB` **variable** (**IV**): The scale expressions for all the derivative terms and the initial condition all simplify to `0.8431`, the magnitude scale factor for the `VP` variable:

```
 -1*fPB    0.8431   = 6.738*9.306e-02*9.966e-02*0.950,14.201
-0.15*VB   0.8431   = 6.738*9.306e-02*0.978*1.630*0.843
 0.5*fPA   0.8431   = 6.738*9.306e-02*6.363e-2*1.812*11.659
```

- `PA` **variable** (**III**): The scale expression for the initial condition and the derivative of the signal both simplify to `7.557`, the magnitude scale factor of the `PA` variable:

```
PA(0)   7.5575   = 8.018*0.943
 PA'    7.5575   = 6.738*1.009*1.113
```

- `PB` **variable**(**III**): The scale expression for the initial condition and the derivative of the signal both simplify to `0.5597`, the magnitude scale factor of the `PB` variable:

```
PB(0)   0.5597   = 0.745*0.751
 PB'    0.5597   = 6.738*9.852e-02*0.843
```

- `PosA` **variable**(**I**): The scale expression for the signal dynamics simplifies to `0.4100`, the magnitude scale factor for the `PosA` variable:

```
PosA   0.4100   = 9.592e-02*0.566*7.557
```

```
var V = integ(0.2*(V*(1.0-U*U)) - U,-0.5);
var U = integ(V,0.0);
var OSC = emit(U);
interval U = [-2.5,2.5];
interval V = [-2.5,2.5];
time 10;
```

Figure 6-24: Dynamical system specification for `vanderpol` benchmark

```
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source V at z; set z0 at 0.000; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source U at z; set z0 at -0.250; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.800; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.000; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block dac @ (0, 3, 0, 0) {
  modes [(const,m)]; set c at 0.200; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source OSC at z; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source V at z0; source V at z1;
  source V at z2; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source U at z1; source U at z2; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source U at z0; source U at z1;
  source U at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
conn block mult port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 3, 0);
conn block dac port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 2, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 0, 1) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 0, 1);
conn block fanout port z1 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 0, 1) with block mult port y loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port y loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 0, 1);
```

Figure 6-25: Unscaled ADP of `vanderpol` benchmark

```
V = integ((((-1.00)*U)+((0.50*((2*0.20)
        +((0.50*((-0.80)*U))*U)))*V)),(2*0))
U = integ((1.00*V),(2*(-0.25)))
OSC = emit((0.60*(1.67*U)))
```

Figure 6-26: Signal dynamics of unscaled ADP for `vanderpol` benchmark

```
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; scale x = 0.247; source V at z; scale z = 0.760; set z0 at 0.000;
  scale z0 = 0.828; }
config block integ @ (0, 3, 1, 0) {
  modes [(h,h,+)]; scale x = 2.584; source U at z; scale z = 7.600; set z0 at -0.250;
  scale z0 = 0.827; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,h,m)]; scale x = 7.600; scale y = 1.000; scale z = 0.247; set c at -1.000;
  scale c = 0.327; }
config block mult @ (0, 3, 1, 1) {
  modes [(m,h,m)]; scale x = 2.666; scale y = 0.760; scale z = 0.247; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(h,m,h)]; scale x = 0.288; scale y = 7.600; scale z = 2.666; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,h,m)]; scale x = 7.600; scale y = 1.000; scale z = 0.288; set c at -0.800;
  scale c = 0.395; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,h)]; scale x = 0.760; scale y = 1.000; scale z = 2.584; set c at 1.000;
  scale c = 0.340; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,h,m)]; scale x = 7.600; scale y = 1.000; scale z = 0.424; set c at 1.667;
  scale c = 0.566; }
config block dac @ (0, 3, 0, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 2.666; set c at 0.200; scale c = 2.902; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.424; source OSC at z; scale z = 0.424; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m)]; scale x = 0.760; source V at z0; scale z0 = 0.760;
  source V at z1; scale z1 = 0.760; source V at z2; scale z2 = 0.760; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,h)]; scale x = 7.600; scale z0 = 7.600; source U at z1; scale z1 = 7.600;
  source U at z2; scale z2 = 7.600; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,h)]; scale x = 7.600; source U at z0; scale z0 = 7.600;
  source U at z1; scale z1 = 7.600; source U at z2; scale z2 = 7.600; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.424; scale z = 0.424; }
conn block mult port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 3, 0);
conn block dac port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 2, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 0, 1) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 0, 1);
conn block fanout port z1 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 0, 1) with block mult port y loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port y loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 0, 1);
timescale 0.317460
```

Figure 6-27: Scaled ADP of `vanderpol` benchmark

```
1  V_sc = (0.7600*V) = integ((3.1500*(0.9754*((0.0995*((0.3273*(-0.9999))*(7.6000*U)))
2           +((0.1221*0.5000)*((((0.9186*2)*(2.9022*0.2000))
3           +((1.2190*0.5000)*((0.0959*((0.3947*(-0.8000))*(7.6000*U)))*(7.6000*U))))*(0.7600*V))))))
4           ,((0.9177*2)*(0.8281*0)))
5  U_sc = (7.6000*U) = integ((3.1500*(0.9337*(10.0058*((0.3398*0.9999)*(0.7600*V))))),
6           ((9.1847*2)*(0.8275*(-0.2500))))
7  OSC_sc = (0.4241*OSC) = (0.6000*emit((0.0987*((0.5656*1.6665)*(7.6000*U)))))
```

Figure 6-28: Scaled signal dynamics for `vanderpol` benchmark

6.3.4 Van der Pol Oscillator (vanderpol)

Figure 6-24 presents the original dynamical system for the vanderpol application. The vanderpol application defines the V and U state variables and the OSC variable:

```
1  V = integ(0.2*(V*(1.0-U*U)) - U,-0.5);
2  U = integ(V,0.0);
3  OSC = emit(U);
```

Unscaled ADP

Figure 6-25 presents the unscaled ADP for the vanderpol application. The ADP has a total of 14 blocks and 17 connections. The circuit contains 6 multipliers, 2 integrators, 1 DAC, 3 current copiers, 1 observation block, and 1 routing (tout) block. Lines 31-47 define the 17 connections necessary to form the desired circuit. The compiler uses the route block to forward the signal to the extout block. The unscaled ADP uses two copiers to produce four copies of U and one copier to produce two copies of V. The ADP uses Kirchoff's law to implement the addition operators in the relation governing V. The circuit instantiates 7 constant data fields to provide the values -1.0, 0.20, -0.80, 0.0, 1.0, -0.25, and 1.67 to the circuit.

Figure 6-26 presents the physics of the of the labelled signals from the unscaled ADP. The ADP implements V and U as analog currents and OSC as an analog voltage:

```
1  V = integ((((-1.00)*U)+((0.50*((2*0.20)
2           +((0.50*((-0.80)*U))*U)))*V)),(2*0))
3  U = integ((1.00*V),(2*(-0.25)))
4  OSC = emit((0.60*(1.67*U)))
```

The dynamics of the U, V, and OSC signals all fail to syntactically the dynamical system dynamic:

- V: The compiler rewrites the 0.2*(V*(1.0-U*U)) term. The 0.2*V and 0.2*U*U*V sub-terms are implemented as 0.5*2*0.20*V and 0.50*0.50*-0.80*U*U*V respectively. The -0.80 coefficient compensates for the two 0.50 device terms introduced by the multipliers. The 0.20 coefficient doesn't need to compensate for any device terms since the 2 and 0.50 terms cancel one another out. The compiler implements the subtraction operation by negating the 0.80 and 1.0 data field values and summing the negated signals. This signal is the analog current at port z of integrator (0,3,3,0).
- U: The compiler implements the initial condition -0.25 as 2*(-0.25) to account for the device term 2 introduced by the integrator. The compiler also introduces the 1.0 data field value into the expression. This signal is the analog current at port z of integrator (0,3,1,0).

- OSC: The OSC signal introduces the 1.67 data field value to correct for the 0.60 device coefficient introduced by the observation block. This signal is the analog voltage at port z of observation block (0,3,2,0).

Scaled ADP

Figure 6-27 presents the scaled adp for the vanderpol benchmark. The scaled ADP has seven data field scale factors, three variable scale factors, and one time scale factor (0.317). The speed of the scaled computation is therefore 0.6349x the baseline integration speed of the device. The compiler also selectively changes the block modes to more effectively scale the circuit. The mode modifications which change block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,h,+)
mult	(0, 3, 1, 0)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 1, 1)	[(m,m,m), (m,h,h), (h,m,h)]	(m,h,m)
mult	(0, 3, 0, 0)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 0, 1)	[(x,m,m), (x,h,h)]	(x,m,h)
mult	(0, 3, 2, 0)	[(x,m,m), (x,h,h)]	(x,h,m)

Each of the above mode changes alters the input-output relation implemented at the output ports of the block. All of the changes to the input-output relation are compensated for by the scaling transform:

- **integrator** (0,3,1,0): The compiler changes the mode from (m,m,+) to (h,h,+). This modification scales the initial value of the output signal by 10 and increases the operating ranges of the block input and output ports.
- **multipliers** (0,3,1,0), (0,3,1,1), (0,3,0,0), **and** (0,3,2,0): The compiler changes the mode from (x,m,m) or (x,h,h) to (x,h,m). This modification scales the output signal by 0.1 and reduces the operating range of the output port (relative to (x,h,h) mode.
- **multiplier** (0,3,0,1): The compiler changes the mode from (x,m,m) or (x,h,h) to (x,m,h). This modification scales the output signal by 10.0 and reduces the operating range of the input port (relative to (x,h,h) mode).

Figure 6-28 presents the dynamics of the scaled scaled signals from the scaled ADP for the vanderpol benchmark:

```
1  V_sc = (0.7600*V) = integ((3.1500*(0.9754*((0.0995*((0.3273*(-0.9999))*(7.6000*U)))
2           +((0.1221*0.5000)*((((0.9186*2)*(2.9022*0.2000))
```

```
3           +((1.2190*0.5000)*((0.0959*((0.3947*(-0.8000))*(7.6000*U)))*(7.6000*U))))*(0.7600*V))))))
4           ,((0.9177*2)*(0.8281*0)))
5   U_sc = (7.6000*U) = integ((3.1500*(0.9337*(10.0058*((0.3398*0.9999)*(0.7600*V))))),
6           ((9.1847*2)*(0.8275*(-0.2500))))
7   OSC_sc = (0.4241*OSC) = (0.6000*emit((0.0987*((0.5656*1.6665)*(7.6000*U)))))
```

The compiler scales the V, U, and OSC variables by 0.7600, 7.600, and 0.424 respectively. I summarize the scaled data field values below:

- In the V signal, the compiler scales the -1.0, 0.20, -0.80, and 0 data field values by 0.327,2.902, 0.395, and 0.828 respectively.
- In the U signal, the compiler scales the 1.0 and -0.25 data field values by 0.340 and 0.827 respectively.
- In the OSC signal, the compiler scales the 1.667 data field value by 0.566.

All compensation terms capture the behavioral deviations found in the device. The compensation terms with values between 0.0918-0.122 also compensate for mode changes which scale signals by 0.1. The 9.185 compensation term also compensates for the mode change which scales the output signal by 10. The compensation terms with values between 0.934-1.219 capture only the behavioral variations present in the device.

Preservation: The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- V **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to 0.760, the magnitude scale factor of the V variable:

```
   -1.0*U      0.7600   =3.150*0.975*9.945e-02*0.327*7.600
    0.20*V     0.7600   = 3.150*0.975*0.122*0.919*2.902*0.760
-0.20*U*U*V    0.7600   = 3.150*0.975*0.122*1.219*9.594e-2*0.395*7.600*7.600*0.760
    V(0)       0.7600   = 0.9177*0.8281
```

- U **variable (III)**: The scale expression for both the initial condition and the derivative of V all simplify to 7.6, the magnitude scale factor of the U variable:

```
    U'     7.600    = 3.150*0.934*10.006*0.340*0.760
   U(0)    7.600    = 9.185*0.827
```

- Pos **variable (I)**:The scale expression for the signal simplifies to 0.424, the magnitude scale factor for the Pos variable:

```
   PosA   0.424    = 9.867e-02*0.566*7.600
```

```
var fD0 = D1-2*D0+2.0
var D0 = integ(1.0*fD0, 0.0)
var fD1 = D0-2*D1+D2
var D1 = integ(1.0*fD1, 0.0)
var fD2 = D1-2*D2+D3
var D2 = integ(1.0*fD2, 0.0)
var fD3 = D2-2*D3
var D3 = integ(1.0*fD3, 0.0)
var POINT = emit(D1)

interval D0 = [0.0,2.0]
interval D1 = [0.0,2.0]
interval D2 = [0.0,2.0]
interval D3 = [0.0,2.0]
time 120
```

Figure 6-29: Dynamical system specification for `heatN4X2` benchmark

```
conn block mult port z loc (0, 3, 2, 1) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 0, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block fanout port x loc (0, 3, 0, 1);
conn block fanout port z0 loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 3, 1);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 3, 0);
conn block fanout port z2 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z2 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z1 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z2 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 2, 0);
conn block dac port z loc (0, 3, 1, 0) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z2 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z0 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 2, 1);
```

Figure 6-30: Connections from unscaled/scaled ADP for `heatN4X2` benchmark

```
config block dac @ (0, 3, 1, 0) {
  modes [(const,m)]; set c at 1.000; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source D0 at z; set z0 at 0.000; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source D1 at z; set z0 at 0.000; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source D2 at z; set z0 at 0.000; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source D3 at z; set z0 at 0.000; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; source fD0 at x; set c at 1.000; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m), (x,h,h)]; source fD1 at x; set c at 1.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; source fD2 at x; set c at 1.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m), (x,h,h)]; source fD3 at x; set c at 1.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source POINT at z; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,-,-,m), (+,-,-,h)]; source D0 at z0; source ((-1)*D0) at z1;
  source ((-1)*D0) at z2; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,-,-,m), (+,-,-,h)]; source ((-1)*D1) at z1; source ((-1)*D1) at z2; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source D1 at z0; source D1 at z1;
  source D1 at z2; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,-,m), (+,+,-,h)]; source D2 at z1; source ((-1)*D2) at z2; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,-,m), (+,+,-,h)]; source D2 at z0; source D2 at z1;
  source ((-1)*D2) at z2; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,-,-,m), (+,-,-,h)]; source D3 at z0; source ((-1)*D3) at z1;
  source ((-1)*D3) at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
```

Figure 6-31: Block Configurations from unscaled ADP for `heatN4X2` benchmark. Refer to Figure 6-30 for connections.

```
fD0 = ((2*1)+(((-1)*D0)+(((-1)*D0)+D1)))
D0 = integ((1.00*fD0),(2*0))
fD1 = (((-1)*D1)+(((-1)*D1)+(D0+D2)))
D1 = integ((1.00*fD1),(2*0))
fD2 = (D1+(((-1)*D2)+(((-1)*D2)+D3)))
D2 = integ((1.00*fD2),(2*0))
fD3 = (D2+(((-1)*D3)+((-1)*D3)))
D3 = integ((1.00*fD3),(2*0))
POINT = emit((0.60*(1.67*D1)))
```

Figure 6-32: Signal dynamics of the unscaled ADP for `heatN4X2` benchmark

```
config block dac @ (0, 3, 1, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 0.475; set c at 1.000; scale c = 0.511; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; scale x = 0.313; source D0 at z; scale z = 0.475; set z0 at 0.000;
  scale z0 = 0.533; }
config block integ @ (0, 3, 1, 0) {
  modes [(h,m,+)]; scale x = 3.202; source D1 at z; scale z = 0.475; set z0 at 0.000;
  scale z0 = 0.513; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; scale x = 0.309; source D2 at z; scale z = 0.475; set z0 at 0.000;
  scale z0 = 0.518; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,m,+)]; scale x = 3.046; source D3 at z; scale z = 0.475; set z0 at 0.000;
  scale z0 = 0.530; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m)]; source fD0 at x; scale x = 0.475; scale y = 1.000; scale z = 0.313;
  set c at 1.000; scale c = 0.680; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,h)]; source fD1 at x; scale x = 0.475; scale y = 1.000; scale z = 3.202;
  set c at 1.000; scale c = 0.688; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m)]; source fD2 at x; scale x = 0.475; scale y = 1.000; scale z = 0.309;
  set c at 1.000; scale c = 0.655; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,h)]; source fD3 at x; scale x = 0.475; scale y = 1.000; scale z = 3.046;
  set c at 1.000; scale c = 0.663; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m)]; scale x = 0.475; scale y = 1.000; scale z = 0.269; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.269; source POINT at z; scale z = 0.269; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,-,-,m)]; scale x = 0.475; source D0 at z0; scale z0 = 0.475;
  source ((-1)*D0) at z1; scale z1 = 0.475; source ((-1)*D0) at z2; scale z2 = 0.475; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,-,-,m)]; scale x = 0.475; scale z0 = 0.475; source ((-1)*D1) at z1; scale z1 = 0.475;
  source ((-1)*D1) at z2; scale z2 = 0.475; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m)]; scale x = 0.475; source D1 at z0; scale z0 = 0.475;
  source D1 at z1; scale z1 = 0.475; source D1 at z2; scale z2 = 0.475; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,-,m)]; scale x = 0.475; scale z0 = 0.475; source D2 at z1; scale z1 = 0.475;
  source ((-1)*D2) at z2; scale z2 = 0.475; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,-,m)]; scale x = 0.475; source D2 at z0; scale z0 = 0.475;
  source D2 at z1; scale z1 = 0.475; source ((-1)*D2) at z2; scale z2 = 0.475; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,-,-,m)]; scale x = 0.475; source D3 at z0; scale z0 = 0.475;
  source ((-1)*D3) at z1; scale z1 = 0.475; source ((-1)*D3) at z2; scale z2 = 0.475; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.269; scale z = 0.269; }
timescale 0.634921
```

Figure 6-33: Block configurations and `timescale` statement from scaled ADP of `heatN4X2` benchmark. Refer to Figure 6-30 for scaled ADP connections.

```
fD0_sc = (0.4750*fD0) = (((0.9292*2)*0.5112)+((0.4750*((-1)*D0))+((0.4750*((-1)*D0))+(0.4750*D1))))
D0_sc = (0.4750*D0) = integ((1.5750*(0.9638*(0.9692*((0.6797*1.0000)*(0.4750*fD0))))),
         ((0.8908*2)*(0.5332*0)))
fD1_sc = (0.4750*fD1) = ((0.4750*((-1)*D1))+((0.4750*((-1)*D1))+((0.4750*D0)+(0.4750*D2))))
D1_sc = (0.4750*D1) = integ((1.5750*(0.0942*(9.7939*((0.6884*1.0000)*(0.4750*fD1))))),
         ((0.9265*2)*(0.5127*0)))
fD2_sc = (0.4750*fD2) = ((0.4750*D1)+((0.4750*((-1)*D2))+((0.4750*((-1)*D2))+(0.4750*D3))))
D2_sc = (0.4750*D2) = integ((1.5750*(0.9754*(0.9939*((0.6549*1.0000)*(0.4750*D2))))),
         ((0.9177*2)*(0.5176*0)))
fD3_sc = (0.4750*fD3) = ((0.4750*D2)+((0.4750*((-1)*D3))+(0.4750*((-1)*D3))))
D3_sc = (0.4750*D3) = integ((1.5750*(0.0990*(9.6717*((0.6630*1.0000)*(0.4750*fD3))))),
         ((0.8967*2)*(0.5297*0)))
POINT_sc = (0.2685*POINT) = (0.6000*emit((0.9996*((0.5655*1.6667)*(0.4750*D1)))))
```

Figure 6-34: Scaled signal dynamics for `heatN4X2` benchmark

6.3.5 Heat Equation (heatN4X2)

Figure 6-29 presents the dynamical system for the heatN4X2 application. The heatN4X2 application defines the fD0, fD1, fD2, fD3, and POINT variables and the D0, D1, D2, and D3 state variables.

```
1  var fD0 = D1-2*D0 + 2.0
2  var D0 = integ(1.0*fD0, 0.0)
3  var fD1 = D0-2*D1+D2
4  var D1 = integ(1.0*fD1, 0.0)
5  var fD2 = D1-2*D2+D3
6  var D2 = integ(1.0*fD2, 0.0)
7  var fD3 = D2-2*D3
8  var D3 = integ(1.0*fD3, 0.0)
9  var POINT = emit(D1)
```

Unscaled ADP

Figure 6-31 and Figure 6-30 presents the unscaled ADP for the heatN4X2 application. The ADP has a total of 18 blocks and 28 connections. The ADP contains 5 multipliers, 4 integrators, 6 current copiers, 1 observation block, and 1 routing (tout) block. The compiler uses the route block to forward the signal implementing POINT to the extout block. The unscaled ADP uses four copiers to produce 2 positive copies and 2 negative copies of the D1 and D2 signals, two copier to produce 1 positive and 2 negative copies of the D0 and D3 signals. The compiler uses the negated copier signals and Kirchhoff's law together to implement the -2*D0, -2*D1, -2*D2, and -2*D3 terms in the dynamical system. The circuit instantiates 9 constant data fields which together provide the values 1.0,0.0,and 1.67 to the circuit.

Figure 6-32 presents the physics of the signals in the unscaled ADP. The ADP implements fD0, D0, fD1, D1, fD2, D2, fD3, and D3 as analog currents and POINT as an analog voltage:

```
1  fD0 = ((2*1)+(((-1)*D0)+(((-1)*D0)+D1)))
2  D0 = integ((1.00*fD0),(2*0))
3  fD1 = (((-1)*D1)+(((-1)*D1)+(D0+D2)))
4  D1 = integ((1.00*fD1),(2*0))
5  fD2 = (D1+(((-1)*D2)+(((-1)*D2)+D3)))
6  D2 = integ((1.00*fD2),(2*0))
7  fD3 = (D2+(((-1)*D3)+((-1)*D3)))
8  D3 = integ((1.00*fD3),(2*0))
9  POINT = emit((0.60*(1.67*D1)))
```

The signal dynamics for the fD0, D0, fD1, D1, fD2, D2, fD3, D3, and POINT signals all do not syntactically match original dynamical system relations:

- fD0: The compiler implements the -2*D0 term with the ((-1)*D0 + (-1)*D0) expression. The compiler implements this expression with two negative copies of the D0 signal and Kirchhoff's law. The compiler implements the 2 term as 2*1 where 2 is a device term and 1 is a data field value. Port x of multiplier (0,3,2,1) implements fD0. This rewritten term requires no constant data fields – the negation is performed by intelligently setting the mode of the copier blocks.

- fD1, fD2, and fD3: The compiler implements the -2*D1, -2*D2, -2*D3 terms with the ((-1)*D0+(-1)*D0),((-1)*D1 +(-1)*D1), ((-1)*D2+(-1)*D2), and ((-1)*D3+(-1)*D3) expressions respectively. The compiler implements these expressions with negative copies of the D1, D2, and D3 signals and Kirchhoff's law. Port x of multiplier (0,3,2,0) implements fD1, port x of multiplier (0,3,1,0) implements fD2, and port x of multiplier (0,3,0,0) implements fD3.

- D0,D1,D2,D3: The signal relations for the state variables largely match the relations from the dynamical system specification. The initial condition does not need to be adjusted because 2*0 is 0. The analog currents at port z of integrators (0,3,2,0),(0,3,1,0), (0,3,3,0), and (0,3,0,0) implement D0, D1, D2, and D3 respectively.

- POINT: The compiler introduces the 1.67 data field to compensate for the 0.60 device term. The signal is the analog voltage at port z of observation block (0,3,2,0).

Scaled ADP

Figure 6-33 and Figure 6-30 presents the scaled adp for the heatN4X2 benchmark. The scaled ADP defines a total of 63 magnitude scaled factors. The scaled ADP specifies nine data field scale factors, nine variable scale factors, and one time scale factor (0.6349). The speed of the computation is 0.6349x the baseline integration speed of the device. The compiler also changes the block modes for a subset of ADP blocks. The block mode changes which alter the input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,m,+)
mult	(0, 3, 2, 0)	[(x,m,m), (x,h,h)]	(x,m,h)
mult	(0, 3, 1, 1)	[(x,m,m), (x,h,h)]	(x,m,h)

- **multipliers**: The compiler changes the mode from `(x,m,m)` or `(x,h,h)` to `(x,m,h)`. This modification enables the compiler to multiply the input signal by a constant value greater than one. This mode change scales the outptu signal by `10` and reduces the operating range of the input port (relative to `(x,h,h)` mode.

- **integrators**: The compiler changes the mode from `(m,m,+)` to `(h,m,+)`. This mode change scales the derivative by `0.1` and increases the operating range of the input port. This mode modification reduces the integration speed of the integrator.

Each of the above mode changes alters the input-output relation implemented at the output ports of the blocks. The scaling transform compensates for these changes to the input-output relation.

Figure 6-34 presents the scaled dynamics of the scaled ADP for the `heatN4X2` benchmark:

```
1  fD0_sc = (0.4750*fD0) = (((0.9292*2)*0.5112)+((0.4750*((-1)*D0))+((0.4750*((-1)*D0))+(0.4750*D1))))
2  D0_sc = (0.4750*D0) = integ((1.5750*(0.9638*(0.9692*((0.6797*1.0000)*(0.4750*fD0))))),
3           ((0.8908*2)*(0.5332*0)))
4  fD1_sc = (0.4750*fD1) = ((0.4750*((-1)*D1))+((0.4750*((-1)*D1))+((0.4750*D0)+(0.4750*D2))))
5  D1_sc = (0.4750*D1) = integ((1.5750*(0.0942*(9.7939*((0.6884*1.0000)*(0.4750*fD1))))),
6           ((0.9265*2)*(0.5127*0)))
7  fD2_sc = (0.4750*fD2) = ((0.4750*D1)+((0.4750*((-1)*D2))+((0.4750*((-1)*D2))+(0.4750*D3))))
8  D2_sc = (0.4750*D2) = integ((1.5750*(0.9754*(0.9939*((0.6549*1.0000)*(0.4750*D2))))),
9           ((0.9177*2)*(0.5176*0)))
10 fD3_sc = (0.4750*fD3) = ((0.4750*D2)+((0.4750*((-1)*D3))+(0.4750*((-1)*D3))))
11 D3_sc = (0.4750*D3) = integ((1.5750*(0.0990*(9.6717*((0.6630*1.0000)*(0.4750*fD3))))),
12          ((0.8967*2)*(0.5297*0)))
13 POINT_sc = (0.2685*POINT) = (0.6000*emit((0.9996*((0.5655*1.6667)*(0.4750*D1)))))
```

The compiler scales the magnitude of the `fD0`, `D0`, `fD1`, `D1`, `fD2`, `D2`, `fD3`, `D3` signals by `0.4750` and the magnitude of the `POINT` signal by `0.2685`. The compiler reports a time scale factor of `0.6349`. The compiler therefore changes the speed of the computation by a factor of $\texttt{1.5750}^{-1}$ or `0.6349`. The scaled signal dynamics use 9 constant data field scale factors, 9 variable scale factors and one time scale factor `0.6349`. I summarize the scaled data field values below:

- `D0`, `D1`, `D2`, and `D3`: For the signals implementing `D0`, `D1`, `D2`, and `D3`, the compiler scales the data field value `1` by `0.6979`, `0.6884`, `0.6549`, and `0.6630` respectively. The compiler scales the `1` data field by different amounts to compensate for variations in the integration speed of the integrators. The initial condition for all the state variables is `0`. The magnitude scale factor applied to the data field `z0` of the integrators therefore has no effect.

- `POINT`: The `1.667` data field value in the `POINT` signal is scaled by `0.5655`.

The compensation terms with values between `0.9292`-`0.9996` capture only the behavioral variations present in the device. The compensation terms with values `0.0942` and `0.0990` capture the mode

changes for modes which introduce 0.1 coefficients into the input-output relation. The compensation terms 9.7939 and 9.6717 capture the mode changes for modes which introduce 10 coefficients into the input-output relation.

Preservation:The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- fD0 **variable (II)**: The scale expression for each of the terms in the sum expression all simplify to 0.4750, the magnitude scale factor of the fD0 variable:

$$\begin{aligned} 2.0 \quad & 0.4750 \quad = 0.9292*0.5112 \\ (-1)*D0 \quad & 0.4750 \quad = 0.4750 \\ (-1)*D0 \quad & 0.4750 \quad = 0.4750 \\ D1 \quad & 0.4750 \quad = 0.4750 \end{aligned}$$

- D0 **variable (III)**: The scale expression for the derivative and the initial condition all simplify to 0.4750, the magnitude scale factor of the D0 variable:

$$\begin{aligned} D0' \quad & 0.4750 \quad = 1.5750*0.9638*0.9692*0.6797*0.4750 \\ D0(0) \quad & 0.4750 \quad = 0.8908*0.5332 \end{aligned}$$

- fD1 **variable (II)**: The scale expression for each of the terms in the sum expression all simplify to 0.4750, the magnitude scale factor of the fD1 variable:

$$\begin{aligned} D0 \quad & 0.4750 \quad = 0.4750 \\ (-1)*D1 \quad & 0.4750 \quad = 0.4750 \\ (-1)*D1 \quad & 0.4750 \quad = 0.4750 \\ D2 \quad & 0.4750 \quad = 0.4750 \end{aligned}$$

- D1 **variable (III)**: The scale expression for the derivative and the initial condition all simplify to 0.4750, the magnitude scale factor of the D1 variable:

$$\begin{aligned} D1' \quad & 0.4750 \quad = 1.5750*0.0942*9.7939*0.6884*0.4750 \\ D1(0) \quad & 0.4750 \quad = 0.9265*0.5127 \end{aligned}$$

- fD2 **variable (II)**: The scale expression for each of the terms in the sum expression all simplify to 0.4750, the magnitude scale factor of the fD2 variable:

```
    D1      0.4750   = 0.4750
(-1)*D2     0.4750   = 0.4750
(-1)*D2     0.4750   = 0.4750
    D3      0.4750   = 0.4750
```

- `D2` **variable (III)**: The scale expression for the derivative and the initial condition all simplify to `0.4750`, the magnitude scale factor of the `D2` variable:

```
   D2'    0.4750   = 1.5750*0.9754*0.9939*0.6549*0.4750
  D2(0)   0.4750   = 0.9177*0.5176
```

- `fD3` **variable (II)**: The scale expression for each of the terms in the sum expression all simplify to `0.4750`, the magnitude scale factor of the `fD3` variable:

```
    D2      0.4750   = 0.4750
(-1)*D3     0.4750   = 0.4750
(-1)*D3     0.4750   = 0.4750
```

- `D3` **variable (III)**: The scale expression for the derivative and the initial condition all simplify to `0.4750`, the magnitude scale factor of the `D3` variable:

```
   D3'    0.4750   = 1.5750*0.0990*9.6717*0.6630*0.4750
  D3(0)   0.4750   = 0.8967*0.5297
```

- `POINT` **variable (I)**:The scale expression for the signal simplifies to `0.2685`, the magnitude scale factor for the `POINT` variable:

```
  PosA   0.2685   = 0.9996*0.5655*0.4750
```

```
var VW = integ(-W, 0.0);
var W = integ(VW, 1.0);
var Y = integ(5*(W + 0.2*(Y*(1.0-X*X)) - X),0.0);
var X = integ(5*(Y),-0.5);
var OSC = emit(X);
interval VW = [-1.0,1.0];
interval W = [-1.0,1.0];
interval X = [-2.2,2.2];
interval Y = [-2.2,2.2];
time 50;
```

Figure 6-35: Dynamical system for `forced` benchmark

```
conn block mult port z loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 3, 1) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 2, 0);
conn block dac port z loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 1);
conn block mult port z loc (0, 3, 1, 1) with block mult port x loc (0, 3, 3, 1);
conn block mult port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 2, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 1, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 1, 0) with block integ port x loc (0, 3, 0, 0);
conn block fanout port z0 loc (0, 3, 1, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z1 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 2, 0) with block mult port y loc (0, 3, 1, 1);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z1 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port y loc (0, 3, 3, 1);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 2, 0);
```

Figure 6-36: Connections from unscaled/scaled ADP for `forced` benchmark

```
VW = integ(((-1)*W),(2*0))
W = integ(VW,(2*0.50))
Y = integ((((0.50*((2*1)+((0.50*((-4)*X))*X)))*Y)+((5*W)+((-5)*X))),(2*0))
X = integ((5*Y),(2*(-0.25)))
OSC = emit((0.60*(1.67*X)))
```

Figure 6-38: Signal dynamics of the unscaled ADP for `forced` benchmark

```
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source W at z; set z0 at 0.500; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source VW at z; set z0 at 0.000; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source Y at z; set z0 at 0.000; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source X at z; set z0 at -0.250; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 5.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -5.000; }
config block mult @ (0, 3, 3, 1) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -4.000; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 5.000; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; set c at 1.000; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source OSC at z; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,-,m), (+,+,-,h)]; source W at z0; source W at z1;
  source ((-1)*W) at z2; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source Y at z0; source Y at z1;
  source Y at z2; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source X at z1; source X at z2; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source X at z0; source X at z1;
  source X at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
```

Figure 6-37: Unscaled ADP for `forced` benchmark

```
VW_sc = (0.4455*VW) = integ((3.2071*(0.0993*(1.3982*((-1)*W)))),((0.8885*2)*(0.5014*0)))
W_sc = (1.3982*W) = integ((3.2071*(0.9786*(0.4455*VW))),((0.7417*2)*(1.8852*0.5000)))
Y_sc = (0.8252*Y) = integ((3.2071*(0.9954*(((1.0920*0.5000)*((((0.9229*2)*0.3108)
        +((11.8046*0.5000)*((0.9927*((0.1575*(-4))*(0.3943*X)))*(0.3943*X))))*(0.8252*Y)))
        +((0.9806*((0.1885*5)*(1.3982*W)))
        +(9.6928*((0.0676*(-5))*(0.3943*X)))))))),((7.8000*2)*(0.1058*0)))
X_sc = (0.3943*X) = integ((3.2071*(0.9619*(0.9723*((0.1593*5)*(0.8252*Y))))),
        ((0.7799*2)*(0.5055*(-0.2500))))
OSC_sc = (0.5182*OSC) = (0.6000*emit((7.0476*((0.1865*1.6665)*(0.3943*X)))))
```

Figure 6-40: Signal dynamics of the scaled ADP for `forced` benchmark

```
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; scale x = 0.446; source W at z; scale z = 1.398; set z0 at 0.500;
  scale z0 = 1.885; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,m,+)]; scale x = 1.398; source VW at z; scale z = 0.446; set z0 at 0.000;
  scale z0 = 0.501; }
config block integ @ (0, 3, 2, 0) {
  modes [(h,h,+)]; scale x = 0.258; source Y at z; scale z = 0.825; set z0 at 0.000;
  scale z0 = 0.106; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; scale x = 0.128; source X at z; scale z = 0.394; set z0 at -0.250;
  scale z0 = 0.506; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m)]; scale x = 1.398; scale y = 1.000; scale z = 0.258; set c at 5.000;
  scale c = 0.189; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,h)]; scale x = 0.394; scale y = 1.000; scale z = 0.258; set c at -5.000;
  scale c = 0.068; }
config block mult @ (0, 3, 3, 1) {
  modes [(m,h,h)]; scale x = 0.287; scale y = 0.825; scale z = 0.258; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(m,m,h)]; scale x = 0.062; scale y = 0.394; scale z = 0.287; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m)]; scale x = 0.394; scale y = 1.000; scale z = 0.062; set c at -4.000;
  scale c = 0.157; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m)]; scale x = 0.825; scale y = 1.000; scale z = 0.128; set c at 5.000;
  scale c = 0.159; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,h)]; scale x = 0.394; scale y = 1.000; scale z = 0.518; set c at 1.667;
  scale c = 0.187; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 0.287; set c at 1.000; scale c = 0.311; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.518; source OSC at z; scale z = 0.518; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,-,m)]; scale x = 1.398; source W at z0; scale z0 = 1.398;
  source W at z1; scale z1 = 1.398; source ((-1)*W) at z2; scale z2 = 1.398; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m)]; scale x = 0.825; source Y at z0; scale z0 = 0.825;
  source Y at z1; scale z1 = 0.825; source Y at z2; scale z2 = 0.825; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m)]; scale x = 0.394; scale z0 = 0.394; source X at z1; scale z1 = 0.394;
  source X at z2; scale z2 = 0.394; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,m)]; scale x = 0.394; source X at z0; scale z0 = 0.394;
  source X at z1; scale z1 = 0.394; source X at z2; scale z2 = 0.394; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.518; scale z = 0.518; }
timescale 0.311811
```

Figure 6-39: Scaled ADP of `forced` benchmark.

6.3.6 Forced Van der Pol Oscillator (forced)

Figure 6-35 presents the original dynamical system for the forced application. The forced application defines VW, W, Y, X state variables and a OSC intermediate variable:

```
1    var VW = integ(-W, 0.0);
2    var W = integ(VW, 1.0);
3    var Y = integ(5*(W + 0.2*(Y*(1.0-X*X)) - X),0.0);
4    var X = integ(5*(Y),-0.5);
5    var OSC = emit(X);
```

Unscaled ADP

Figure 6-37 and Figure 6-36 present the unscaled ADP for the forced application. The ADP has a total of 18 blocks and 22 connections. The circuit contains 7 multipliers, 4 integrators, 4 current copiers, 1 observation block, and 1 route (tout) block. The compiler uses the tout route block to forward the signal implementing OSC to the extout block. The compiler uses 2 copiers to produce 4 copies of dynamical system variable X, 1 copier to produce 2 copies of variable Y, and one copier to produce 2 copies of variable W. The ADP uses Kirchhoff's law to implement the addition operations in the relation governing Y. The circuit instantiates 10 constant data fields which provide the 0.50, 0.0, 1.0, -4.0, 5, -5, -0.25, and 1.67 values to the circuit.

Figure 6-38 presents the dynamics of the signals from the unscaled ADP for the forced benchmark. The ADP implements the W, VW, Y, and X signals as analog currents and the OSC signal as an analog voltage:

```
1  VW = integ(((-1)*W),(2*0))
2  W = integ(VW,(2*0.50))
3  Y = integ((((0.50*((2*1)+((0.50*((-4)*X))*X)))*Y)+((5*W)+((-5)*X))),(2*0))
4  X = integ((5*Y),(2*(-0.25)))
5  OSC = emit((0.60*(1.67*X)))
```

The W, VW, Y, X, and OSC signal expressions syntactically do not match the original dynamical system relations:

- VW: The compiler implements the -W term as (-1)*W. The compiler produces the negated signal (-1)*W by strategically setting a current copier mode. The initial condition 0.0 is implemented as 2*0 where 0 is a data field value and 2 is a device termin introduced by an integrator block. The analog current at port z of integrator (0,3,0,0) implements VW.

- `W`: The compiler implements the `1` initial condition as `2*0.50` where `0.50` is a data field value `2` is a device term introduced by the integrator block. The analog current at port `z` of integrator `(0,3,1,0)` implements `W`.
- `Y`: The compiler distributes the `5` coefficient to the `-X`, `W`, and `0.2*(Y*(1.0+X*X))` terms in the derivative. The compiler also changes the `-X*X` term to `-4*X*X` to compensate for the two `0.5` device terms introduced by the multipliers in the ADP. The analog current at port `z` of integrator `(0,3,2,0)` implements `Y`.
- `X`: The compiler implements the `-0.5` initial condition as `2*0.25` where `0.25` is a data field value and `2.0` is a device term introduced by the integrator block. The analog current at port `z` of integrator `(0,3,3,0)` implements `X`.
- `OSC`: The compiler introduces the `1.67` data field value to compensate for the `0.60` device term introduced by the observation block. The analog voltage at port `z` of observation block `(0,3,2,0)` implements `OSC`.

Scaled ADP

Figure 6-39 and Figure 6-36 present the scaled ADP for the `forced` application. The scaled ADP defines a total of 63 magnitude scale factors. The scaled ADP specifies 10 data field scale factors, 5 variable scale factors, and one time scale factor (`0.3118`). The speed of the scaled computation is therefore 0.3118x the baseline integration speed of the device. The compiler also changes the block modes for a subset of ADP blocks to better scale the circuit. The block mode changes which alter the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 2, 0)	[(m,m,+)]	(h,h,+)
mult	(0, 3, 1, 0)	[(x,m,m), (x,h,h)]	(x,m,h)
mult	(0, 3, 1, 1)	[(m,m,m), (m,h,h), (h,m,h)]	(m,m,h)
mult	(0, 3, 2, 1)	[(x,m,m), (x,h,h)]	(x,m,h)

- **integrator** `(0,3,0,0)`: The compiler changes the mode from `(m,m,+)` to `(h,m,+)`. This modification scales the derivative of the output signal by `0.1` and increases the operating range of the block input port `x`.
- **integrator** `(0,3,2,0)`: The compiler changes the mode from `(m,m,+)` to `(h,h,+)`. This modification scales the initial value of the output signal by `10` and increases the operating range of the block input port `x` and block output port `z`.

- **multipliers** `(0,3,1,0)` **and** `(0,3,2,1)`: The compiler changes the mode from `(x,m,m)` or `(x,h,h)` to `(x,m,h)`. This modification scales the output signal by `10` and decreases the operating range of the input port `x` (relative to mode `(x,h,h)`).
- **multiplier** `(0,3,1,1)`: The compiler changes the mode from `(m,m,m)`, `(m,h,h)`, or `(h,m,h)` to `(m,m,h)`. This scales the output signal by `10` and decreases the operating range of either the `x` or the `y` input ports.

Each of the above mode changes alters the input-output relations implemented by the blocks. The scaling transform compensates for these changes to the input-output relation.

Figure 6-40 presents the scaled dynamics of the scaled ADP for the `forced` benchmark.

```
1  VW_sc = (0.4455*VW) = integ((3.2071*(0.0993*(1.3982*((-1)*W)))),((0.8885*2)*(0.5014*0)))
2  W_sc = (1.3982*W) = integ((3.2071*(0.9786*(0.4455*VW))),((0.7417*2)*(1.8852*0.5000)))
3  Y_sc = (0.8252*Y) = integ((3.2071*(0.9954*(((1.0920*0.5000)*((((0.9229*2)*0.3108)
4           +((11.8046*0.5000)*((0.9927*((0.1575*(-4))*(0.3943*X)))*(0.3943*X))))*(0.8252*Y)))
5           +((0.9806*((0.1885*5)*(1.3982*W)))
6           +(9.6928*((0.0676*(-5))*(0.3943*X)))))))),((7.8000*2)*(0.1058*0)))
7  X_sc = (0.3943*X) = integ((3.2071*(0.9619*(0.9723*((0.1593*5)*(0.8252*Y))))),
8           ((0.7799*2)*(0.5055*(-0.2500))))
9  OSC_sc = (0.5182*OSC) = (0.6000*emit((7.0476*((0.1865*1.6665)*(0.3943*X)))))
```

The compiler scales the magnitude of the `VW`, `W`, `Y`, `X`, and `OSC` signals by `0.4455`, `1.3982`, `0.8252`, `0.3943`, and `0.5182` respectively. The compiler reports a time scale factor of `0.3118`. The compiler therefore changes the speed of the computation by a factor of $\texttt{3.2071}^{-1}$ or `0.3118`. The scaled dynamics use 10 constant data field scale factors, 5 variable scale factors and one time scale factor `0.3118`. I summarize the scaled data fields below:

- In the `W` and `VW` signals, the compiler scales the `0.50` and `0.0` data field values by `1.8852` and `0.5014` respectively.
- In the `Y` signal, the compiler scales the`-4`, `5`, `-5`, and `0` data field values by `0.1575`, `0.1885`, `0.0676`, and `0.1058` respectively.
- In the `OSC` and `X` signals, the compiler scales the `5`, `-0.25`, and `1.67` data field values by `0.1593`, `0.5055`, and `0.1865` respectively.

The compensation terms with values between `0.7417`-`0.9954` capture only the behavioral variations present in the device. All other compensation terms capture both the effect of changing the block mode and the behavioral deviations found in the device. The compensation term `0.0993` compensates for a mode change which introduces the `0.1` coefficient into the block input-output relation. The compensation terms with values between `7.8`-`11.8046` compensate for mode changes introduce the `10` coefficient into the block input-output relation.

Preservation:The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- `VW` **variable (III)**: The scale expression for both the initial condition and the derivative of `VW` all simplify to `0.4455`, the magnitude scale factor of the `VW` variable:

```
VW'    0.4455   3.2071*0.0993*1.3982
VW(0)  0.4455   0.8885*0.5014
```

- `W` **variable (III)**: The scale expression for both the initial condition and the derivative of `W` all simplify to `1.3982`, the magnitude scale factor of the `W` variable:

```
W'    1.3982   = 3.2071*0.9786*0.4455
W(0)  1.3982   = 0.7417*1.8852
```

- `Y` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to `0.8252`, the magnitude scale factor of the `Y` variable:

```
(-5.0)*(X)   0.8252   = 3.2071*0.9954*9.6928*0.0676*0.3943
(-5.0)*(W)   0.8252   = 3.2071*0.99540.9806*0.1885*1.3982
    Y        0.8252   = 3.2071*0.9954*1.0920*0.8252*0.9229*0.3108
-(X*X)*Y     0.8252   = 3.2071*0.9954*1.0920*0.8252*11.8046*0.9927*0.1575
                        *0.3943*0.3943
  Y(0)       0.8252   = 7.8000*0.1058
```

- `X` **variable (III)**: The scale expression for both the initial condition and the derivative of `X` all simplify to `0.3943`, the magnitude scale factor of the `X` variable:

```
X'    0.3943   =3.2071*0.9619*0.9723*0.1593*0.8252
X(0)  0.3943   =0.7799*0.5055
```

- `OSC` **variable (I)**:The scale expression for the signal simplifies to `0.5182`, the magnitude scale factor for the `OSC` variable:

```
OSC   0.5182   = 7.0476*0.1865*0.3943
```

```
var VSIG = integ(-0.25*VSIG, 0.0);
var SIG = integ(VSIG, 1.0);
var ERR = PLANT-SIG;
var CTRL =-2.0*ERR-8.0*INTEG;
var INTEG = integ(ERR - 0.3*INTEG,0.0);
var PLANT = integ(CTRL + 1, 0.0)
var Controlled = emit(CTRL);

interval VSIG = [-0.5,0.5];
interval SIG = [-1.0,1.0];
interval PLANT = [-2.0,2.0];
interval CTRL = [-2.0,2.0];
interval ERR = [-2.0,2.0];
interval INTEG = [-2.0,2.0];
time 200;
```

Figure 6-41: Dynamical system specification for `pid` benchmark

```
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 2, 1) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 1, 0);
conn block dac port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z1 loc (0, 3, 3, 0) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z1 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z0 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 3, 1);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z0 loc (0, 3, 3, 1) with block integ port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 3, 1) with block mult port x loc (0, 3, 0, 1);
conn block mult port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block mult port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 3, 1);
conn block mult port z loc (0, 3, 3, 1) with block fanout port x loc (0, 3, 3, 1);
```

Figure 6-42: Connections from unscaled/scaled ADP for `pid` benchmark

```
VSIG = integ(((-0.25)*SIG),(2*0))
SIG = integ(VSIG,(2*0.50))
ERR = (((-1.00)*SIG)+PLANT)
CTRL = (((-2)*ERR)+((-8)*INTEG))
INTEG = integ(((1.00*ERR)
        +((-0.30)*INTEG)),(2*0))
PLANT = integ(((2*0.50)+CTRL),(2*0))
Controlled = emit((0.60*(1.67*CTRL)))
```

Figure 6-44: Signal dynamics of the unscaled ADP for `pid` benchmark

```
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source SIG at z; set z0 at 0.500; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source VSIG at z; set z0 at 0.000; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source INTEG at z; set z0 at 0.000; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source PLANT at z; set z0 at 0.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.250; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -2.000; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -8.000; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -0.300; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; set c at 0.500; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source Controlled at z; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source SIG at z0; source SIG at z1;
  source SIG at z2; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source ERR at x; source ERR at z0;
  source ERR at z1; source ERR at z2; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source CTRL at x; source CTRL at z0;
  source CTRL at z1; source CTRL at z2; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source INTEG at z0; source INTEG at z1;
  source INTEG at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
```

Figure 6-43: Unscaled ADP for `pid` benchmark

```
VSIG_sc = (3.1333*VSIG) = integ((3.7462*(0.0993*(0.7227*((1.0293*(-0.2500))*(11.3168*SIG))))),
        ((0.8885*2)*(3.5263*0)))
SIG_sc = (11.3168*SIG) = integ((3.7462*(0.9641*(3.1333*VSIG))),((6.0031*2)*(1.8852*0.5000)))
ERR_sc = (3.5427*ERR) = ((0.9708*((0.3225*(-1.0000))*(11.3168*SIG)))
        +(3.5427*PLANT))
CTRL_sc = (0.9500*CTRL) = ((0.9854*((0.2721*(-2))*(3.5427*ERR)))
        +(9.6822*((0.1188*(-8))*(0.8263*INTEG))))
INTEG_sc = (0.8263*INTEG) = integ((3.7462*(0.0945*((0.6991*((0.9426*1.0000)*(3.5427*ERR)))
        +(0.9658*((2.9253*(-0.3000))*(0.8263*INTEG)))))),((0.5922*2)*(1.3951*0)))
PLANT_sc = (3.5427*PLANT) = integ((3.7462*(0.9954*(((0.9229*2)*(1.0294*0.5000))
        +(0.9500*CTRL)))),((7.8000*2)*(0.4542*0)))
Controlled_sc = (0.5334*Controlled) = (0.6000*emit((0.9927*((0.5656*1.6667)*(0.9500*CTRL)))))
```

Figure 6-46: Scaled signal dynamics for `pid` benchmark

```
config block integ @ (0, 3, 3, 0) {
  modes [(h,h,+)]; scale x = 3.133; source SIG at z; scale z = 11.317; set z0 at 0.500;
  scale z0 = 1.885; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,m,+)]; scale x = 8.419; source VSIG at z; scale z = 3.133; set z0 at 0.000;
  scale z0 = 3.526; }
config block integ @ (0, 3, 1, 0) {
  modes [(h,m,+)]; scale x = 2.334; source INTEG at z; scale z = 0.826;
  set z0 at 0.000; scale z0 = 1.395; }
config block integ @ (0, 3, 2, 0) {
  modes [(h,h,+)]; scale x = 0.950; source PLANT at z; scale z = 3.543;
  set z0 at 0.000; scale z0 = 0.454; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,h,h)]; scale x = 11.317; scale y = 1.000; scale z = 8.419; set c at -0.250;
  scale c = 1.029; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,h,h)]; scale x = 11.317; scale y = 1.000; scale z = 3.543; set c at -1.000;
  scale c = 0.322; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,h,h)]; scale x = 3.543; scale y = 1.000; scale z = 0.950; set c at -2.000;
  scale c = 0.272; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,h)]; scale x = 0.826; scale y = 1.000; scale z = 0.950; set c at -8.000;
  scale c = 0.119; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,h,h)]; scale x = 3.543; scale y = 1.000; scale z = 2.334; set c at 1.000;
  scale c = 0.943; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m)]; scale x = 0.826; scale y = 1.000; scale z = 2.334; set c at -0.300;
  scale c = 2.925; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m)]; scale x = 0.950; scale y = 1.000; scale z = 0.533; set c at 1.667;
  scale c = 0.566; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 0.950; set c at 0.500; scale c = 1.029; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.533; source Controlled at z; scale z = 0.533; }
config block fanout @ (0, 3, 3, 0) {
  modes [(+,+,+,h)]; scale x = 11.317; source SIG at z0; scale z0 = 11.317;
  source SIG at z1; scale z1 = 11.317; source SIG at z2; scale z2 = 11.317; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,+,h)]; source ERR at x; scale x = 3.543; source ERR at z0;
  scale z0 = 3.543; source ERR at z1; scale z1 = 3.543; source ERR at z2;
  scale z2 = 3.543; }
config block fanout @ (0, 3, 3, 1) {
  modes [(+,+,+,m)]; source CTRL at x; scale x = 0.950; source CTRL at z0;
  scale z0 = 0.950; source CTRL at z1; scale z1 = 0.950; source CTRL at z2;
  scale z2 = 0.950; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,m)]; scale x = 0.826; source INTEG at z0; scale z0 = 0.826;
  source INTEG at z1; scale z1 = 0.826; source INTEG at z2; scale z2 = 0.826; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.533; scale z = 0.533; }
timescale 0.266937
```

Figure 6-45: Scaled ADP for `pid` benchmark.

6.3.7 PID Controller (pid)

Figure 6-41 presents the original dynamical system for the pid application. The pid application defines VSIG, SIG, INTEG, PLANT state variables and ERR, CTRL, and Controlled intermediate variables:

```
1  var VSIG = integ(-0.25*VSIG, 0.0);
2  var SIG = integ(VSIG, 1.0);
3  var ERR = PLANT-SIG;
4  var CTRL = -2.0*ERR-8.0*INTEG;
5  var INTEG = integ(ERR - 0.3*INTEG,0.0);
6  var PLANT = integ(CTRL + 1, 0.0)
7  var Controlled = emit(CTRL);
```

Unscaled ADP

Figure 6-43 and Figure 6-42 present the unscaled ADP for the pid application. The ADP has a total of 18 blocks and 21 connections. The circuit contains 7 multipliers, 4 integrators, 1 DAC, 4 current copiers, 1 observation block, and 1 route (tout) block. The compiler uses the tout route block to forward the signal implementing CTRL to the extout block. The unscaled ADP uses four copiers to produce 2 copies of the SIG, ERR, CTRL, and INTEG signals. The compiler uses Kirchhoff's law to implement the addition operators in the relations governing ERR, CTRL, INTEG, and PLANT variables. The circuit instantiates 12 constant data fields to provide the values 0.50, -0.25, 0.0, -1.0, -2.0, -8.0, 1.0, -0.30, 0.50, and 1.67 to the circuit.

Figure 6-44 presents the dynamics of the signals from the unscaled ADP for the pid benchmark. The ADP implements the VSIG, SIG, INTEG, PLANT, ERR, and CTRL signals as analog currents and the Controlled signal as an analog voltage:

```
1  VSIG = integ(((-0.25)*SIG),(2*0))
2  SIG = integ(VSIG,(2*0.50))
3  ERR = (((-1.00)*SIG)+PLANT)
4  CTRL = (((-2)*ERR)+((-8)*INTEG))
5  INTEG = integ(((1.00*ERR)+((-0.30)*INTEG)),(2*0))
6  PLANT = integ(((2*0.50)+CTRL),(2*0))
7  Controlled = emit((0.60*(1.67*CTRL)))
```

The SIG, ERR, CTRL, INTEG, PLANT, and Controlled signal expressions do not syntactically match the original dynamical system relations:

- VSIG: The compiler implements the initial condition 0.0 as 2*0.0 where 0.0 is a data field value and 2 is a device term introduced by an integrator block. The analog current at port z of integrator (0,3,0,0) implements the VSIG variable.

- SIG: The compiler implements the initial condition from 1.0 as 2*0.5 where 0.50 to account for the device term 2.0. The analog current at port z of integrator (0,3,3,0) implements the SIG variable.

- ERR: The compiler negates the SIG variable by introducing the -1.0 data field value. The compiler then adds the (-1)*SIG signal to the PLANT signal. This is necessary because subtraction is not explicitly supported in the device. The analog current at port x of current copier (0,3,2,1) implements the ERR variable.

- CTRL: The compiler implements -2*ERR-8*INTEG expression by multiplying ERR and INTEG by 2 and -8 respectively and then adding the terms together. This is necessary because subtraction is not explicitly supported in the device. The analog current at port x of current copier (0,3,3,1) implements the CTRL variable.

- INTEG: The compiler implements the ERR term as 1.00*ERR, where 1.00 is a data field value. The compiler implements the ERR-0.3*INTEG expression by multiplying ERR and INTEG by 1.0 and -0.3 and then summing the terms together. Again, the compiler performs this transformation because subtraction is not supported in the device. The 1.0 and -0.3 coefficients are implemented as data field values. The initial condition 0.0 is implemented as 2*0.0 where 0.0 is a data field value and 2 is a device term introduced by the integrator block. The analog current at port z of integrator (0,3,1,0) implements the INTEG variable.

- PLANT: The compiler implements 1.0 term as 2*0.5, where 0.5 is a data field value and 2 is a device constant introduced by the DAC. The analog current at port z of integrator (0,3,2,0) implements the PLANT variable.

- Controlled: The compiler introduces the 1.67 data field value to compensate for the 0.60 device term introduced by the observation block. The analog current at port z of observation block (0,3,2,0) implements the Controlled variable.

Scaled ADP

Figure 6-46 and Figure 6-42 present the scaled dynamics of the scaled ADP for the pid benchmark. The scaled ADP has a total of 63 magnitude scale factors. The scaled ADP has 12 data field scale factors, 7 variable scale factors, and one time scale factor (0.2669). The speed of the scaled computation is 0.2669x the baseline integration speed of the device. The compiler also changes the

block modes for a subset of the ADP blocks to better scale the circuit. The block mode changes which alter the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 3, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 2, 0)	[(m,m,+)]	(h,h,+)
mult	(0, 3, 3, 1)	[(x,m,m), (x,h,h)]	(x,m,h)

- **integrators** `(0,3,3,0)` **and** `(0,3,2,0)`: The compiler changes the integrator mode from `(m,m,+)` to `(h,h,+)`. This modification to the mode scales the initial value of the signal by `10` and increases the operating range of the input port `x` and output port `z` of the block.
- **integrators** `(0,3,0,0)` **and** `(0,3,1,0)`: The compiler changes the integrator mode from `(m,m,+)` to `(h,m,+)`. This modification scales the derivative of the output signal by `0.1` and increases the operating range of the input port `x`.
- **multiplier** `(0,3,3,1)`: The compiler changes the multiplier mode from `(x,m,m)` or `(x,h,h)` to `(x,m,h)`. This modification scales the output signal by `10.0` and reduces the operating range of the input port `x` (relative to mode `(x,h,h)`).

Each of the above mode changes alters the input-output relations implemented by the block. The scaling transform compensates for these changes to the input-output relation.

Figure 6-46 presents the scaled dynamics of the scaled ADP for the `pid` benchmark:

```
1  VSIG_sc = (3.1333*VSIG) = integ((3.7462*(0.0993*(0.7227*((1.0293*(-0.2500))*(11.3168*SIG))))),
2          ((0.8885*2)*(3.5263*0)))
3  SIG_sc = (11.3168*SIG) = integ((3.7462*(0.9641*(3.1333*VSIG))),((6.0031*2)*(1.8852*0.5000)))
4  ERR_sc = (3.5427*ERR) = ((0.9708*((0.3225*(-1.0000))*(11.3168*SIG)))
5          +(3.5427*PLANT))
6  CTRL_sc = (0.9500*CTRL) = ((0.9854*((0.2721*(-2))*(3.5427*ERR)))
7          +(9.6822*((0.1188*(-8))*(0.8263*INTEG))))
8  INTEG_sc = (0.8263*INTEG) = integ((3.7462*(0.0945*((0.6991*((0.9426*1.0000)*(3.5427*ERR)))
9          +(0.9658*((2.9253*(-0.3000))*(0.8263*INTEG))))))),((0.5922*2)*(1.3951*0)))
10 PLANT_sc = (3.5427*PLANT) = integ((3.7462*(0.9954*(((0.9229*2)*(1.0294*0.5000))
11         +(0.9500*CTRL)))),((7.8000*2)*(0.4542*0)))
12 Controlled_sc = (0.5334*Controlled) = (0.6000*emit((0.9927*((0.5656*1.6667)*(0.9500*CTRL)))))
```

The compiler scales the `VSIG`, `SIG`, `ERR`, `CTRL`, `INTEG`, `PLANT`, and `Controlled` variables by `3.1333`, `11.3166`, `3.5427`, `0.9500`, `0.8263`, `3.5427`, and `0.5335` respectively. The compiler reports a time scale factor of `0.2669`. The compiler therefore scales the speed of the computation by a factor

of 3.7462^{-1} or 0.2669. The scaled ADP has 12 data field scale factors, 7 variable scale factors, and one time scale factor. I summarize the scaled data field values below:

- In the scaled VSIG and SIG signals, the compiler scales the data field values -0.25, 0, and 0.50 by 1.0293,3.5263, and 1.8852 respectively.
- In the scaled ERR,CTRL,INTEG signals, the compiler scales the -2, -8, 1.0, -0.3, and 0.0 data field values by 0.2721, 0.1188, 0.9426, 2.9253, and 1.3951 receptively.
- In the scaled PLANT and Controlled signals, the compiler scales the 0.5, 0.0, and 1.667 data field by 1.0294, 0.4542, and 0.5656 respectively.

The compensation terms with values between 0.5922-0.9954 capture only behavioral variations in the device. The remaining compensation terms capture both the effect of changing the block mode and model the behavioral deviations found in the device. The compensation terms with the values 0.0993 and 0.0945 capture the mode changes which introduce the constant coefficient 0.1 into the block input-output relations. The compensation terms between 6.00-9.68 capture the mode changes which introduce the constant coefficient 10 into the block input-output relations.

Preservation: The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- VSIG **variable (III)**: The scale expression for both the initial condition and the derivative of VSIG all simplify to 3.1333, the magnitude scale factor of the VSIG variable:

$$\begin{aligned} \text{VSIG'} \quad 3.1333 &= 3.7462*0.0993*0.7227*1.0293*11.3168*\text{SIG} \\ \text{VSIG(0)} \quad 3.1333 &= 0.8885*3.5263 \end{aligned}$$

- SIG **variable (III)**: The scale expression for both the initial condition and the derivative of SIG all simplify to 11.3168, the magnitude scale factor of the SIG variable:

$$\begin{aligned} \text{SIG'} \quad 11.3168 &= 3.7462*0.9641*3.1333 \\ \text{SIG(0)} \quad 11.3168 &= 6.0031*1.8852 \end{aligned}$$

- ERR **variable (II)**: The scale expression for each of the terms in the signal expression for ERR all simplify to 3.5427, the magnitude scale factor of the ERR variable:

$$\begin{aligned} \text{-SIG} \quad 3.5427 &= 0.9708*0.3225*11.3168 \\ \text{PLANT} \quad 3.5427 &= 3.5427 \end{aligned}$$

- `CTRL` **variable (II)**: The scale expression for each of the terms in the signal expression for `CTRL` all simplify to `0.9500`, the magnitude scale factor of the `CTRL` variable:

```
-2.0*ERR    0.9500  = 0.9854*0.2721*3.5427
-8.0*INTEG  0.9500  = 9.6822*0.1188*0.8263
```

- `INTEG` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to `0.8263`, the magnitude scale factor of the `INTEG` variable:

```
   ERR      0.8263  = 3.7462*0.0945*0.6991*0.9426*3.5427
-0.3*INTEG  0.8263  = 3.7462*0.0945*0.9658*2.9253*0.8263
 INTEG(0)   0.8263  = 0.5922*1.3951
```

- `PLANT` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to `3.5427`, the magnitude scale factor of the `PLANT` variable:

```
   1.0      3.5427  = 3.7462*0.9954*0.9229*1.0294
  CTRL      3.5427  = 3.7462*0.9954*0.9500
PLANT(0)    3.5427  = 7.8000*0.4542
```

- `Controlled` **variable (I)**:The scale expression for the signal simplifies to `0.5334`, the magnitude scale factor for the `Controlled` variable:

```
Controlled  0.5334  =0.9927*0.5656*0.9500
```

```
var VSIG = integ(-0.04*SIG, 0.0)
var SIG = integ(VSIG, 0.7)
var E = SIG-X
var RP = (1/2.0)*P
var X = integ(RP*E, 0)
var P = integ(0.6-RP*P, 0.0)
var STATE = emit(X)
interval VSIG = [-0.3,0.3]
interval SIG = [-1.0,1.0]
interval X = [-1.0,1.0]
interval P = [0.0,1.0]
time 50
```

Figure 6-47: Dynamical system specification for `kalman` benchmark

```
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source SIG at z; set z0 at 0.350; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source VSIG at z; set z0 at 0.000; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source X at z; set z0 at 0.000; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source P at z; set z0 at 0.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.040; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m), (x,h,h)]; source RP at z; set c at 0.500; }
config block mult @ (0, 3, 0, 1) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; source E at x; set c at 2.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -2.000; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; set c at 0.300; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source STATE at z; }
config block fanout @ (0, 3, 1, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source SIG at z0; source SIG at z1;
  source SIG at z2; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source RP at z0; source RP at z1;
  source RP at z2; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,-,m), (+,+,-,h)]; source X at z0; source X at z1;
  source ((-1)*X) at z2; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source P at z0; source P at z1;
  source P at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 0) with block mult port x loc (0, 3, 0, 1);
conn block mult port z loc (0, 3, 3, 1) with block mult port x loc (0, 3, 0, 0);
conn block dac port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 1, 1) with block fanout port x loc (0, 3, 1, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 1, 1) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 1);
conn block fanout port z0 loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 1);
conn block fanout port z1 loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 1, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 1, 0);
```

Figure 6-48: Unscaled ADP for `kalman` benchmark

```
VSIG = integ(((-0.04)*SIG),(2*0))
SIG = integ(VSIG,(2*0.35))
E = (SIG+((-1)*X))
RP = (0.50*P)
X = integ(((0.50*(2.00*(SIG+((-1)*X))))*RP),(2*0))
P = integ(((2*0.30)+((0.50*((-2)*P))*RP)),(2*0))
STATE = emit((0.60*(1.67*X)))
```

Figure 6-49: Signal dynamics of the unscaled ADP for `kalman` benchmark

```
config block integ @ (0, 3, 1, 0) {
  modes [(h,h,+)]; scale x = 3.756; source SIG at z; scale z = 11.176; set z0 at 0.350;
  scale z0 = 1.351; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,m,+)]; scale x = 12.002; source VSIG at z; scale z = 3.756;
  set z0 at 0.000; scale z0 = 4.227; }
config block integ @ (0, 3, 3, 0) {
  modes [(h,h,+)]; scale x = 3.680; source X at z; scale z = 11.176; set z0 at 0.000;
  scale z0 = 1.862; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; scale x = 0.614; source P at z; scale z = 1.900; set z0 at 0.000;
  scale z0 = 2.382; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,h,m)]; scale x = 11.176; scale y = 1.000; scale z = 12.002; set c at -0.040;
  scale c = 10.932; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m)]; scale x = 1.900; scale y = 1.000; source RP at z; scale z = 2.103;
  set c at 0.500; scale c = 1.146; }
config block mult @ (0, 3, 0, 0) {
  modes [(m,m,h)]; scale x = 0.140; scale y = 2.103; scale z = 3.680; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,h,m)]; source E at x; scale x = 11.176; scale y = 1.000; scale z = 0.140;
  set c at 2.000; scale c = 0.128; }
config block mult @ (0, 3, 0, 1) {
  modes [(m,m,m)]; scale x = 0.214; scale y = 2.103; scale z = 0.614; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m)]; scale x = 1.900; scale y = 1.000; scale z = 0.214; set c at -2.000;
  scale c = 0.115; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,h,h)]; scale x = 11.176; scale y = 1.000; scale z = 1.140; set c at 1.667;
  scale c = 0.146; }
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 0.614; set c at 0.300; scale c = 0.665; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 1.140; source STATE at z; scale z = 1.140; }
config block fanout @ (0, 3, 1, 1) {
  modes [(+,+,+,h)]; scale x = 11.176; source SIG at z0; scale z0 = 11.176;
  source SIG at z1; scale z1 = 11.176; source SIG at z2; scale z2 = 11.176; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,+,m)]; scale x = 2.103; source RP at z0; scale z0 = 2.103;
  source RP at z1; scale z1 = 2.103; source RP at z2; scale z2 = 2.103; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,-,h)]; scale x = 11.176; source X at z0; scale z0 = 11.176;
  source X at z1; scale z1 = 11.176; source ((-1)*X) at z2; scale z2 = 11.176; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m)]; scale x = 1.900; source P at z0; scale z0 = 1.900;
  source P at z1; scale z1 = 1.900; source P at z2; scale z2 = 1.900; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 1.140; scale z = 1.140; }
conn block mult port z loc (0, 3, 3, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 1, 0) with block mult port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 3, 0) with block mult port x loc (0, 3, 0, 1);
conn block dac port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 1, 1);
conn block mult port z loc (0, 3, 1, 1) with block fanout port x loc (0, 3, 1, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 2, 1);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 1, 1) with block mult port x loc (0, 3, 3, 1);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z1 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z0 loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 0);
conn block fanout port z1 loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 1);
conn block fanout port z0 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 1, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z2 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 1, 0);
timescale 0.317460
```

Figure 6-50: Scaled ADP of `kalman` benchmark

```
VSIG_sc = (3.7561*VSIG) = integ((3.1500*(0.0993*(0.0982*((10.9319*(-0.0400))*(11.1765*SIG))))),
          ((0.8885*2)*(4.2272*0)))
SIG_sc = (11.1765*SIG) = integ((3.1500*(0.9446*(3.7561*VSIG))),((8.2713*2)*(1.3512*0.3500)))
E_sc = (11.1765*E) = ((11.1765*SIG)+(11.1765*((-1)*X)))
RP_sc = (2.1034*RP) = (0.9658*((1.1462*0.5000)*(1.9000*P)))
X_sc = (11.1765*X) = integ((3.1500*(0.9641*((12.4843*0.5000)*((0.0977*((0.1283*1.9998)*((11.1765*SIG)
          +(11.1765*((-1)*X)))))*(2.1034*RP))))),
          ((6.0031*2)*(1.8618*0)))
P_sc = (1.9000*P) = integ((3.1500*(0.9821*(((0.9229*2)*(0.6655*0.3000))
          +((1.3613*0.5000)*((0.9806*((0.1151*(-2))*(1.9000*P)))*(2.1034*RP)))))),((0.7978*2)*(2.3815*0)))
STATE_sc = (1.1401*STATE) = (0.6000*emit((0.6991*((0.1459*1.6665)*(11.1765*X)))))
```

Figure 6-51: Scaled signal dynamics for `kalman` benchmark

6.3.8 Kalman Filter (kalman)

Figure 6-47 presents the original dynamical system for the kalman application. The kalman application definfes VSIG, SIG, X, and P state variables and E and RP intermediate variables:

```
1    var VSIG = integ(-0.04*SIG, 0.0)
2    var SIG = integ(VSIG, 0.7)
3    var E = SIG-X
4    var RP = 0.50*P
5    var X = integ(RP*E, 0)
6    var P = integ(0.6-RP*P, 0.0)
7    var STATE = emit(X)
```

Unscaled ADP

Figure 6-48 presents the unscaled ADP for the kalman application. The ADP has a total of 18 blocks and 21 connections. The circuit contains 7 multipliers, 4 integrators, 1 DAC, 4 current copiers, 1 observation block, and 1 route (tout) block. The compiler uses the tout route block to forward the signal to the extout block. The unscaled ADP uses 4 copier to produce 2 copies of the SIG, RP, X, and P signals. The ADP uses Kirchhoff's law to implement the addition operators in the relation governing E and P. The circuit instantiates 10 constant data fields which provide the 0.35, -0.04, 0.50, 2.0, 0.30, -2.0, 1.67, and 0 values to the circuit.

Figure 6-49 presents the dynamics of the signals from the unscaled ADP for the kalman benchmark. The ADP implements VSIG, SIG E, RP, X, and P variables with analog currents and the STATE variable with an analog voltage:

```
1  VSIG = integ(((-0.04)*SIG),(2*0))
2  SIG = integ(VSIG,(2*0.35))
3  E = (SIG+((-1)*X))
4  RP = (0.50*P)
5  X = integ(((0.50*(2.00*E))*RP),(2*0))
6  P = integ(((2*0.30)+((0.50*((-2)*P))*RP)),(2*0))
7  STATE = emit((0.60*(1.67*X)))
```

The SIG, E, X, P, and STATE signal expressions do not syntactically match the original dynamical system relations. I summarize the relationship between the dynamical system relations and the signal dynamics below:

- VSIG: The compiler implements the initial condition 0.0 as 2*0.0 where 0.0 is a data field value and 2 is a device term introduced by the integrator. The analog current at port z of

integrator (0,3,0,0) implements the VSIG variable.

- SIG: The compiler implements the initial condition 0.7 as 2*0.35 where 0.35 is a data field value and 2 is a device term introduced by the integrator block. The analog current at port z of integrator (0,3,1,0) implements the SIG variable.
- E: The compiler implements the expression SIG-X by negating X and then adding it with SIG. This is necessary because subtraction is not explicitly supported in the device. Note that this negation operation is done without introducing a constant data field value. The analog current at port x of multiplier (0,3,1,0) implements the E variable.
- RP: The signal expression which implements RP matches the dynamical system relation for RP. The analog current at port z of multiplier (0,3,1,1) implements the RP variable.
- X: The compiler implmements RP*E as 0.50*2.00*E*RP. The compiler introduces the 2.00 data field value to compensate for the out the 0.50 device term introduced by the signal multiplier block. The initial condition 0.0 is implemented as 2*0.0 where 0.0 is a data field value and 2 is a device term. The analog current at port z of integrator (0,3,3,0) implements the X variable.
- P: The compiler introduces the 2.00 data field value to compensate for the 0.50 device constant introduced by the multiplier block. The compiler implements the 0.6 term as 2*0.30 where 0.30 is a data field value and 2.0 is the device constant introduced by the DAC. The compiler implements the subtraction by negating RP*P and adding it to 2*0.30. The analog current at port z of the integrator (0,3,2,0) implements the P variable.
- STATE: The compiler introduces the 1.67 data field value to compensate for the 0.60 device term introduced by the observation block. The analog voltage at port z of the observation block at (0,3,2,0) implements the STATE variable.

Scaled ADP

Figure 6-50 presents the scaled ADP for the kalman benchmark. The scaled ADP has a total of 63 magnitude scale factors. The scaled ADP has 10 data field scale factors, 7 variable scale factors, and one time scale factor (0.3174). The speed of the scaled computation is therefore 0.3174x the baseline integration speed of the device. The compiler also changes th eblock mode for a subset of ADP blocks. The block mode modifications which change the input-output relations implemented by the blocks are summarized below:

- **multipliers (0,3,3,1) and (0,3,1,0)**: The compiler changes the block mode from (x,m,m) or (x,h,h) to (x,h,m). This modification scales the output signal by 0.1 and reduces the operating range of the output port z (relative to the (x,h,h) mode).

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 3, 0)	[(m,m,+)]	(h,h,+)
mult	(0, 3, 3, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 0, 0)	[(m,m,m), (m,h,h), (h,m,h)]	(m,m,h)
mult	(0, 3, 1, 0)	[(x,m,m), (x,h,h)]	(x,h,m)

- **multiplier** (0,3,0,0): The compiler changes the block mode from (m,m,m), (m,h,h) or (h,m,h) to (m,m,h). This modification scales the output signal by 10.0 and reduces the operating range of the either the input port x or the input port y (relative to mode (m,h,h) or (h,m,h)).
- **integrators** (0,3,1,0) **and** (0,3,3,0): The compiler changes the block mode from (m,m,+) to (h,h,+). This modification scales the initial condition by 10 and increases the operating range of the input port x and the output port z.
- **integrator** (0,3,0,0): The compiler changes the block mode from (m,m,+) to (h,m,+). This modification scales the derivative of the signal by 0.1 and increases the operating range of the input port x.

Each of the above mode modifications changes the input-output relations implemented by the block. The scaling transform compensates for these changes to the input-output relations.

Figure 6-51 presents the scaled dynamics of the scaled ADP for the kalman benchmark.

```
1  VSIG_sc = (3.7561*VSIG) = integ((3.1500*(0.0993*(0.0982*((10.9319*(-0.0400))*(11.1765*SIG))))),
2          ((0.8885*2)*(4.2272*0)))
3  SIG_sc = (11.1765*SIG) = integ((3.1500*(0.9446*(3.7561*VSIG))),
4          ((8.2713*2)*(1.3512*0.3500)))
5  E_sc = (11.1765*E) = ((11.1765*SIG))+((11.1765*((-1)*X)))
6  RP_sc = (2.1034*RP) = (0.9658*((1.1462*0.5000)*(1.9000*P)))
7  X_sc = (11.1765*X) = integ((3.1500*(0.9641
8          *((12.4843*0.5000)*((0.0977*((0.1283*1.9998)*(11.1765*E)))*(2.1034*RP))))),
9          ((6.0031*2)*(1.8618*0)))
10 P_sc = (1.9000*P) = integ((3.1500*(0.9821*(((0.9229*2)*(0.6655*0.3000)))
11         +(((1.3613*0.5000)*((0.9806*((0.1151*(-2))*(1.9000*P)))*(2.1034*RP)))))),
12         ((0.7978*2)*(2.3815*0)))
13 STATE_sc = (1.1401*STATE) = (0.6000*emit((0.6991*((0.1459*1.6665)*(11.1765*X)))))
```

The compiler scales the VSIG, SIG, E, RP, X, P, and STATE variables by 3.7561, 11.1764, 11.1765, 2.1034, 11.1765, 1.900, and 1.1401 respectively. The compiler reports a time scale factor of 0.3174. The compiler therefore changes the speed of the computation by a factor of 3.1500^{-1} or 0.3174. The scaled signal dynamics use 10 data field scale factors, 7 variable scale factors, and one time scale factor 0.3174. I summarize the scaled data field values below:

- In the VSIG and SIG signals, the compiler scales the 0.04, 0.0, and 0.35 data field values by 10.9319,4.2272, and 1.3512 respectively.
- In the RP and X signals, the compiler scales the 0.5, 1.9998 and 0.0 data field values by 1.1462, 0.1283, and 1.8618 respectively.
- In the P and STATE signals, the compiler scales the 0.3, -2, 0, and 1.67 data field values by 0.6655, 0.1151, 2.3815, and 0.1459 respectively.

The compensation terms with values between 0.6991-9806 capture the behavioral variations present in the device. All other compensation terms capture both the effects of changing the block mode and the behavioral deviations present in the device. The compensation terms with values between 0.0977-0.0993 capture mode modifications which scale signals by 0.1. The compensation terms 6.0031 and 8.2713 capture mode modifications which scale signals by 10.

Preservation: The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- VSIG **variable (III)**: The scale expression for both the initial condition and the derivative of VSIG all simplify to 3.7561, the magnitude scale factor of the VSIG variable:

$$\begin{aligned} \text{VSIG'} \quad 3.7561 &= 0.0993*0.0982*10.9319*11.1765 \\ \text{VSIG(0)} \quad 3.7561 &= 0.8885*4.2272 \end{aligned}$$

- SIG **variable (III)**: The scale expression for both the initial condition and the derivative of SIG all simplify to 11.1765, the magnitude scale factor of the SIG variable:

$$\begin{aligned} \text{SIG'} \quad 11.1765 &= 0.9446*3.7561 \\ \text{SIG(0)} \quad 11.1765 &= 8.2713*1.3512 \end{aligned}$$

- E **variable (II)**: The scale expression for each of the terms in the signal expression for E all simplify to 11.1765, the magnitude scale factor of the E variable. Note that because this signal contains no data fields and therefore has no additional degrees of freedom, all terms are scaled by the magnitude scale factor for E:

$$\begin{aligned} \text{SIG} \quad 11.1765 &= 11.1765 \\ \text{-X} \quad 11.1765 &= 11.1765 \end{aligned}$$

- RP **variable (I)**:The scale expression for the signal simplifies to 2.1034, the magnitude scale factor for the RP variable:

$$\text{RP} \quad 2.1034 = 0.9658*1.1462*1.9000$$

- `X` **variable (III)**: The scale expression for both the initial condition and the derivative of `X` all simplify to `11.1765`, the magnitude scale factor of the `X` variable:

```
  X'    11.1765   = 3.1500*0.9641*12.4843*0.0977*0.1283*11.1765*2.1034
 X(0)   11.1765   = 6.0031*1.8618
```

- `P` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition for the signal `P` all simplify to `1.900`, the magnitude scale factor of the `P` variable:

```
  0.6    1.900   = 3.1500*0.9821*0.9229*0.6655
 -RP*P   1.900   = 3.1500*0.9821*1.3613*0.9806*0.1151*1.9000*2.1034
 P(0)    1.900   = 0.7978*2.3815
```

- `STATE` **variable (I)**:The scale expression for the signal simplifies to `1.1401`, the magnitude scale factor for the `STATE` variable:

```
STATE   1.1401   = 0.6991*0.1459*11.1765
```

```
  var E = 0.8-ES
  var S = 0.5-ES
  var ES = integ(E*S - 0.3*ES, 0.0)
  var COMPLEX = emit(ES)
  interval E = [0.0,0.8]
  interval S = [0.0,0.5]
  interval ES = [0.0,0.5]
  time 20
```

Figure 6-52: Dynamical system specification for `smmrxn` benchmark

```
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; set c at 0.400; }
config block dac @ (0, 3, 1, 0) {
  modes [(const,m)]; set c at 0.250; }
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source ES at z; set z0 at 0.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.300; }
config block mult @ (0, 3, 0, 0) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; source S at y; set c at 0.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; source E at x; set c at 2.000; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source COMPLEX at z; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,-,m), (+,+,-,h)]; source ES at z1; source ((-1)*ES) at z2; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,-,m), (+,+,-,h)]; source ES at z0; source ES at z1;
  source ((-1)*ES) at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
conn block mult port z loc (0, 3, 0, 1) with block mult port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 1);
conn block dac port z loc (0, 3, 2, 0) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z2 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 0, 1);
conn block dac port z loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 0);
conn block fanout port z2 loc (0, 3, 2, 0) with block mult port y loc (0, 3, 0, 0);
```

Figure 6-53: Unscaled ADP for `smmrxn` benchmark

```
E = ((2*0.40)+((-1)*ES))
S = ((2*0.25)+((-1)*ES))
ES = integ((((-0.30)*ES)+((0.50*(2.00*((2*0.40)
        +((-1)*ES))))*((2*0.25)+((-1)*ES)))),(2*0))
COMPLEX = emit((0.60*(1.67*ES)))
```

Figure 6-54: Signal dynamics of the unscaled ADP for `smmrxn` benchmark

```
config block dac @ (0, 3, 2, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 2.175; set c at 0.400; scale c = 2.356; }
config block dac @ (0, 3, 1, 0) {
  modes [(const,m)]; scale x = 1.000; scale z = 2.175; set c at 0.250; scale c = 2.345; }
config block integ @ (0, 3, 2, 0) {
  modes [(h,m,+)]; scale x = 6.768; source ES at z; scale z = 2.175; set z0 at 0.000;
  scale z0 = 3.012; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m)]; scale x = 2.175; scale y = 1.000; scale z = 6.768; set c at -0.300;
  scale c = 3.167; }
config block mult @ (0, 3, 0, 0) {
  modes [(m,m,h)]; scale x = 0.249; source S at y; scale y = 2.175; scale z = 6.768;
  set c at 0.000; scale c = 1.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m)]; source E at x; scale x = 2.175; scale y = 1.000; scale z = 0.249;
  set c at 2.000; scale c = 0.115; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m)]; scale x = 2.175; scale y = 1.000; scale z = 1.206; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 1.206; source COMPLEX at z; scale z = 1.206; }
config block fanout @ (0, 3, 2, 1) {
  modes [(+,+,-,m)]; scale x = 2.175; scale z0 = 2.175; source ES at z1; scale z1 = 2.175;
  source ((-1)*ES) at z2; scale z2 = 2.175; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,-,m)]; scale x = 2.175; source ES at z0; scale z0 = 2.175;
  source ES at z1; scale z1 = 2.175; source ((-1)*ES) at z2; scale z2 = 2.175; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 1.206; scale z = 1.206; }
conn block mult port z loc (0, 3, 0, 1) with block mult port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 3, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 2, 1) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 1);
conn block fanout port z1 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 1);
conn block dac port z loc (0, 3, 2, 0) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z2 loc (0, 3, 2, 1) with block mult port x loc (0, 3, 0, 1);
conn block dac port z loc (0, 3, 1, 0) with block mult port y loc (0, 3, 0, 0);
conn block fanout port z2 loc (0, 3, 2, 0) with block mult port y loc (0, 3, 0, 0);
timescale 0.316998
```

Figure 6-55: Scaled ADP of `smmrxn` benchmark.

```
E_sc = (2.1747*E) = (((0.9229*2)*(2.3564*0.4000))
        +(2.1747*((-1)*ES)))
S_sc = (2.1747*S) = (((0.9275*2)*(2.3448*0.2500))
        +(2.1747*((-1)*ES)))
ES_sc = (2.1747*ES) = integ((3.1546*(0.1019*((0.9827*((3.1667*(-0.3000))*(2.1747*ES)))
        +((12.4843*0.5000)*((0.9927*((0.1155*1.9998)*(((0.9229*2)*(2.3564*0.4000))
        +(2.1747*((-1)*ES)))))*(((0.9275*2)*(2.3448*0.2500))
        +(2.1747*((-1)*ES))))))),((0.7219*2)*(3.0124*0)))
COMPLEX_sc = (1.2062*COMPLEX) = (0.6000*emit((0.9806*((0.5656*1.6665)*(2.1747*ES)))))
```

Figure 6-56: Scaled signal dynamics for `smmrxn` benchmark

6.3.9 Michaelis Menten Reaction (smmrxn)

Figure 6-52 presents the original dynamical system for the smmrxn application. The smmrxn application defines E, S, and ES state variables and a COMPLEX intermediate variable:

```
1 var E = 0.8-ES
2 var S = 0.5-ES
3 var ES = integ(E*S - 0.3*ES, 0.0)
4 var COMPLEX = emit(ES)
```

Unscaled ADP

Figure 6-53 presents the unscaled ADP for the smmrxn application. The ADP has a total of 11 blocks and 13 connections. The circuit contains 4 multipliers, one integrator, 2 DACs, 2 current copiers, 1 observation block, and one route (tout) block. The compiler uses the tout route block to forward the signal to the extout block. The unscaled ADP uses 2 copiers to produce 4 copies of ES. The compiler uses Kirchhoff's law to implement the addition operators in the relations governing E,S, and ES. The circuit instantiates 6 constant data fields to provide the 0.40,0.25,-0.30,2.00,1, and 1.67 values to the circuit.

Figure 6-54 presents the dynamics of the signals from the unscaled ADP for the smmrxn benchmark. The unscaled ADP implements the E, S, and ES variables as analog currents and the COMPLEX variable as an analog voltage:

```
1 E = ((2*0.40)+((-1)*ES))
2 S = ((2*0.25)+((-1)*ES))
3 ES = integ((((-0.30)*ES)+((0.50*(2.00*(E))*(S)),(2*0))
4 COMPLEX = emit((0.60*(1.67*ES)))
```

The E, S, ES, and COMPLEX signal expressions do not syntactically match the original dynamical system relations:

- E: The compiler implements the 0.8 term as 2*0.4 where 0.4 is a data field value and 2.0 is a device term introduced introduced by a DAC block. The compiler implements the subtraction operation by negating ES and adding it to 2*0.4. The analog current at port x of multiplier (0,3,0,1) implements the E variable.
- S: The compiler implements the 0.5 term as 2*0.25 where 0.25 is a data field value and 2.0 is a device term introduced by a DAC block. The compiler implements the subtraction operation bu negating ES and adding it to 2*0.25. The analog current at port y of multiplier (0,3,0,0) implements the S variable.

- ES: The compiler introduces the 2.00 data field value to compensate for the device term 0.5 which was introduced by the multiplier. The compiler implements subtraction by negating 0.30*ES and adding the negated signal to E*S. The analog current at port z of integrator (0,3,2,0) implements the ES variable.
- COMPLEX: The compiler introduces the 1.67 data field value to cancel out the 0.60 device term introduced by the observation block. The analog voltage at port z of observation block (0,3,2,0) implements the COMPLEX variable.

Scaled ADP

Figure 6-55 presents the scaled ADP for the smmrxn benchmark. The scaled ADP defines a total of 37 magnitude scale factors. The scaled ADP specifies 6 data field scale factors, 4 variable scale factors, and one time scale factor (0.3170). The speed of the scaled computation is 0.3170x the baseline integration speed of the device. The compiler also changes the block modes for a subset of the ADP blocks. The block mode modifications which alter the block's input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 2, 0)	[(m,m,+)]	(h,m,+)
mult	(0, 3, 0, 0)	[(m,m,m), (m,h,h), (h,m,h)]	(m,m,h)

- **multiplier** (0,3,0,0): The compiler changes the mode from (m,m,m), (m,h,h), or (h,m,h) to (m,m,h). This modification scales the output signal by 10.0 and reduces the operating range of either input port x or y (relative to mode (h,m,h) and (m,h,h)).
- **integrator** (0,3,2,0): The compiler changes the mode from (m,m,+) to (h,m,+). This modification scales the derivative of the output signal by 0.1 and expands the operating range of the input port x.

Each of the above mode changes alters the input-output relation implemented at the output ports of the block. The scaling transform compensates for these changes in the behavior of each block.

Figure 6-56 presents the scaled dynamics of the scaled ADP for the smmrxn benchmark.

```
1  E_sc = (2.1747*E) = (((0.9229*2)*(2.3564*0.4000))
2          +(2.1747*((-1)*ES)))
3  S_sc = (2.1747*S) = (((0.9275*2)*(2.3448*0.2500))
4          +(2.1747*((-1)*ES)))
5  ES_sc = (2.1747*ES) = integ((3.1546*(0.1019*((0.9827*((3.1667*(-0.3000))*(2.1747*ES)))
6          +((12.4843*0.5000)*((0.9927*((0.1155*1.9998)*(2.1747*E)*(2.1747*S), ((0.7219*2)*(3.0124*0)))
7  COMPLEX_sc = (1.2062*COMPLEX) = (0.6000*emit((0.9806*((0.5656*1.6665)*(2.1747*ES)))))
```

The compiler scales the E, S, and ES variables by 2.1747 and the COMPLEX variable by 1.2062. The compiler reports a time scale factor of 0.3170. The compiler therefore scales the speed of the computation by a factor of 0.3170^{-1} or 3.1546. The scaled signal dynamics use 6 data field scale factors, 4 variable scale factors, and one time scale factor. I summarize the scaled data field values below:

- In the E and S signals, the compiler scales the 0.40 and 0.25 data field value by 2.3564 and 2.3448 respectively.
- In the ES signal, the -0.3, 2.0, and 0 data field values are scaled by 3.1667, 0.1155, and 3.0124 respectively.
- In the COMPLEX signal, the 1.67 data field value is scaled by 0.5656.

The compensation terms with values between 0.7219-0.9806 capture the behavioral variations present in the device. All other compensation terms capture both the behavioral deviations within the device and the effects of modifying the mode on the input-output relation. The compensation term 0.1019 captures the 0.1 coefficient introduced by a change of mode. The compensation term 12.4843 captures the 10.0 coefficient introduced by a change of mode.

Preservation: The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- E **variable (II)**: The scale expression for each of the terms in the signal expression for E all simplify to 2.1747, the magnitude scale factor of the E variable:

$$\begin{aligned} 0.8 \quad 2.1747 &= 0.9229*2.3564 \\ ES \quad 2.1747 &= 2.1747 \end{aligned}$$

- S **variable (II)**: The scale expression for each of the terms in the signal expression for 2.1747 all simplify to S, the magnitude scale factor of the S variable:

$$\begin{aligned} 0.5 \quad 2.1747 &= 0.9275*2.3448 \\ -ES \quad 2.1747 &= 2.1747 \end{aligned}$$

- ES **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to 2.1747, the magnitude scale factor of the ES variable:

$$\begin{aligned} -ES \quad 2.1747 &= 3.1546*0.1019*0.9827*3.1667*2.1747 \\ E*S \quad 2.1747 &= 3.1546*0.1019*12.4843*0.9927*0.1155*2.1747*2.1747 \\ ES(0) \quad 2.1747 &= 0.7219*3.0124 \end{aligned}$$

- `COMPLEX` **variable (I)**:The scale expression for the signal simplifies to `1.2062`, the magnitude scale factor for the `COMPLEX` variable:

```
COMPLEX   1.2062   = 0.9806*0.5656*2.1747
```

```
func utf(T) = 15.62/pow((1+max(T,0)),2.5)
func vtf(T) = 15.6/(1+max(T,0))
func umod(T) = pow((1+max(T,0)),-2.0015)

var VPERT = integ(-PERT, 0.0)
var PERT = integ(VPERT, 0.5)
var IPTG = 0.5+PERT
var UMOD = call(umod,IPTG)
var UTF = call(utf,V)
var VTF = call(vtf,(U*UMOD))
var V = integ(VTF - V, 0.0)
var U = integ(UTF - U, 0.0)
var compV = emit(V)



interval UMOD = [0.0,1.0]
interval UTF = [0.0,16.0]
interval VTF = [0.0,16.0]
interval V = [0.0,16.0]
interval U = [0.0,1.2]
time 20
```

Figure 6-57: Dynamical system specification for `gentog` benchmark

```
conn block mult port z loc (0, 2, 1, 1) with block integ port x loc (0, 2, 1, 0);
conn block adc port z loc (0, 2, 2, 0) with block lut port x loc (0, 2, 2, 0);
conn block lut port z loc (0, 2, 2, 0) with block dac port x loc (0, 2, 2, 0);
conn block adc port z loc (0, 3, 0, 0) with block lut port x loc (0, 3, 0, 0);
conn block lut port z loc (0, 3, 0, 0) with block dac port x loc (0, 3, 0, 0);
conn block adc port z loc (0, 3, 2, 0) with block lut port x loc (0, 3, 2, 0);
conn block lut port z loc (0, 3, 2, 0) with block dac port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 2, 0) with block adc port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 2, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 0, 0) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 2, 3, 0) with block fanout port x loc (0, 2, 2, 0);
conn block integ port z loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 0, 1);
conn block integ port z loc (0, 2, 1, 0) with block integ port x loc (0, 2, 3, 0);
conn block fanout port z0 loc (0, 2, 2, 0) with block mult port x loc (0, 2, 1, 1);
conn block fanout port z0 loc (0, 3, 0, 0) with block adc port x loc (0, 3, 0, 0);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z2 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 0);
conn block dac port z loc (0, 2, 2, 0) with block tout port x loc (0, 2, 0, 0);
conn block tout port z loc (0, 2, 0, 0) with block tin port x loc (0, 3, 0, 0);
conn block tin port z loc (0, 3, 0, 0) with block mult port y loc (0, 3, 2, 0);
conn block fanout port z0 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z2 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 1);
conn block dac port z loc (0, 3, 2, 0) with block mult port x loc (0, 3, 2, 1);
conn block dac port z loc (0, 3, 0, 0) with block mult port x loc (0, 3, 1, 0);
conn block dac port z loc (0, 2, 3, 0) with block adc port x loc (0, 2, 2, 0);
conn block fanout port z1 loc (0, 2, 2, 0) with block adc port x loc (0, 2, 2, 0);
```

Figure 6-58: Connections from unscaled/scaled ADP for `gentog` benchmark

```
config block integ @ (0, 2, 3, 0) {
  modes [(m,m,+)]; source PERT at z; set z0 at 0.250; }
config block integ @ (0, 2, 1, 0) {
  modes [(m,m,+)]; source VPERT at z; set z0 at 0.000; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source V at z; set z0 at 0.000; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source U at z; set z0 at 0.000; }
config block mult @ (0, 2, 1, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 3, 2, 0) {
  modes [(m,m,m), (m,h,h), (h,m,h)]; set c at 0.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.000; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -1.000; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.000; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block dac @ (0, 2, 3, 0) {
  modes [(const,m)]; set c at 0.250; }
config block dac @ (0, 2, 2, 0) {
  modes [(dyn,h)]; source UMOD at z; set c at 0.000; }
config block dac @ (0, 3, 0, 0) {
  modes [(dyn,h)]; source UTF at z; set c at 0.000; }
config block dac @ (0, 3, 2, 0) {
  modes [(dyn,h)]; source VTF at z; set c at 0.000; }
config block lut @ (0, 2, 2, 0) {
  modes [(*)]; set e at (1*(0.05*pow((1)+(max((2*(1*y)),0)),(-2.00)))); }
config block lut @ (0, 3, 0, 0) {
  modes [(*)]; set e at (1*(0.05*(15.62*pow((1)+(pow(max((2*(1*y)),0),2.50)),(-1))))); }
config block lut @ (0, 3, 2, 0) {
  modes [(*)]; set e at (1*(0.05*(15.60*pow((1)+(max((2*(1*y)),0)),(-1))))); }
config block adc @ (0, 2, 2, 0) {
  modes [(m)]; source IPTG at x; }
config block adc @ (0, 3, 0, 0) {
  modes [(m)]; }
config block adc @ (0, 3, 2, 0) {
  modes [(m)]; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source compV at z; }
config block fanout @ (0, 2, 2, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source PERT at z0; source PERT at z1;
  source PERT at z2; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source V at z0; source V at z1;
  source V at z2; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source U at z0; source U at z1;
  source U at z2; }
config block tout @ (0, 3, 0, 0) { modes [(*)]; }
config block tout @ (0, 2, 0, 0) { modes [(*)]; }
config block tin @ (0, 3, 0, 0) { modes [(*)]; }
```

Figure 6-59: Block Configurations from unscaled ADP for `gentog` benchmark. Refer to Figure 6-58 for scaled ADP connections.

```
VPERT = integ(((-1.00)*PERT),(2*0))
PERT = integ(VPERT,(2*0.25))
IPTG = ((2*0.25)+PERT)
UMOD = (20*(0.05*pow((1+max((2*(0.50*IPTG)),0)),(-2.00))))
UTF = (20*(0.05*(15.62*pow((1+pow(max((2*(0.50*V)),0),2.50)),(-1)))))
VTF = (20*(0.05*(15.60*pow((1+max((2*(0.50*((0.50*(U+U))*UMOD))),0)),(-1)))))
V = integ((((-1.00)*V)+(1.00*VTF)),(2*0))
U = integ((((-1.00)*U)+(1.00*UTF)),(2*0))
compV = emit((0.60*(1.67*V)))
```

Figure 6-60: Signal dynamics of the unscaled ADP for `gentog` benchmark

```
config block integ @ (0, 2, 3, 0) {
  modes [(h,h,+)]; scale x = 1.660; source PERT at z; scale z = 18.102;
  set z0 at 0.250; scale z0 = 2.653; }
config block integ @ (0, 2, 1, 0) {
  modes [(h,m,+)]; scale x = 1.431; source VPERT at z; scale z = 1.660;
  set z0 at 0.000; scale z0 = 2.146; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,h,+)]; scale x = 0.098; source V at z; scale z = 1.107; set z0 at 0.000;
  scale z0 = 0.141; }
config block integ @ (0, 3, 1, 0) {
  modes [(h,h,+)]; scale x = 0.639; source U at z; scale z = 6.918; set z0 at 0.000;
  scale z0 = 0.836; }
config block mult @ (0, 2, 1, 1) {
  modes [(x,h,m)]; scale x = 18.102; scale y = 1.000; scale z = 1.431; set c at -1.000;
  scale c = 0.791; }
config block mult @ (0, 3, 2, 0) {
  modes [(m,h,h)]; scale x = 6.918; scale y = 1.753; scale z = 14.967; set c at 0.000;
  scale c = 1.000; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,h,m)]; scale x = 1.107; scale y = 1.000; scale z = 0.098; set c at -1.000;
  scale c = 0.896; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m)]; scale x = 0.110; scale y = 1.000; scale z = 0.098; set c at 1.000;
  scale c = 0.943; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,h,m)]; scale x = 6.918; scale y = 1.000; scale z = 0.639; set c at -1.000;
  scale c = 0.950; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,h)]; scale x = 0.111; scale y = 1.000; scale z = 0.639; set c at 1.000;
  scale c = 0.595; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,h,m)]; scale x = 1.107; scale y = 1.000; scale z = 0.060; set c at 1.667;
  scale c = 0.566; }
config block dac @ (0, 2, 3, 0) {
  modes [(const,h)]; scale x = 1.000; scale z = 18.102; set c at 0.250; scale c = 1.958; }
config block dac @ (0, 2, 2, 0) {
  modes [(dyn,m)]; scale x = 18.272; source UMOD at z; scale z = 1.753; set c at 0.000;
  scale c = 1.000; }
config block dac @ (0, 3, 0, 0) {
  modes [(dyn,m)]; scale x = 1.207; source UTF at z; scale z = 0.111; set c at 0.000;
  scale c = 1.000; }
config block dac @ (0, 3, 2, 0) {
  modes [(dyn,m)]; scale x = 1.186; source VTF at z; scale z = 0.110; set c at 0.000;
  scale c = 1.000; }
config block lut @ (0, 2, 2, 0) {
  modes [(*)]; scale x = 1.885; scale z = 18.272;
  set e at (18.2724*(0.0500*pow((1)+(max((2*(0.5305*y)),0)),(-2.0015)))); scale e = 18.272; }
config block lut @ (0, 3, 0, 0) {
  modes [(*)]; scale x = 0.118; scale z = 1.207;
  set e at (1.2069*(0.0500*(15.6200*pow((1)+(pow(max((2*(8.4874*y)),0),2.5000)),(-1))))); scale e = 1.207; }
config block lut @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 1.571; scale z = 1.186;
  set e at (1.1860*(0.0500*(15.6000*pow((1)+(max((2*(0.6366*y)),0)),(-1))))); scale e = 1.186; }
config block adc @ (0, 2, 2, 0) {
  modes [(h)]; source IPTG at x; scale x = 18.102; scale z = 1.885; }
config block adc @ (0, 3, 0, 0) {
  modes [(h)]; scale x = 1.107; scale z = 0.118; }
config block adc @ (0, 3, 2, 0) {
  modes [(h)]; scale x = 14.967; scale z = 1.571; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 0.060; source compV at z; scale z = 0.060; }
config block fanout @ (0, 2, 2, 0) {
  modes [(+,+,+,h)]; scale x = 18.102; source PERT at z0; scale z0 = 18.102;
  source PERT at z1; scale z1 = 18.102; source PERT at z2; scale z2 = 18.102; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,h)]; scale x = 1.107; source V at z0; scale z0 = 1.107;
  source V at z1; scale z1 = 1.107; source V at z2; scale z2 = 1.107; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,h)]; scale x = 6.918; source U at z0; scale z0 = 6.918;
  source U at z1; scale z1 = 6.918; source U at z2; scale z2 = 6.918; }
config block tout @ (0, 3, 0, 0) { modes [(*)]; scale x = 0.060; scale z = 0.060; }
config block tout @ (0, 2, 0, 0) { modes [(*)]; scale x = 1.753; scale z = 1.753; }
config block tin @ (0, 3, 0, 0) { modes [(*)]; scale x = 1.753; scale z = 1.753; }
timescale 0.087259
```

Figure 6-61: Block configurations and `timescale` statement from scaled ADP of `gentog` benchmark. Refer to Figure 6-58 for scaled ADP connections.

```
VPERT_sc = (1.6604*VPERT) = integ((11.4602*(0.1012*(0.1000*((0.7909*(-1.0000))*(18.1021*PERT))))),
        ((0.7738*2)*(2.1458*0)))
PERT_sc = (18.1021*PERT) = integ((11.4602*(0.9513*(1.6604*VPERT))),((6.8244*2)*(2.6526*0.2500)))
IPTG_sc = (18.1021*IPTG) = (((9.2446*2)*(1.9581*0.2500))+(18.1021*PERT))
UMOD_sc = (1.7527*UMOD) = ((0.0959*20)*(18.2724*(0.0500*pow((1+max((2*(0.5305
        *((0.1041*0.5000)*(18.1021*IPTG)))),0)),(-2.0015)))))
UTF_sc = (0.1109*UTF) = ((0.0919*20)*(1.2069*(0.0500*(15.6200*pow((1+pow(max((2*(8.4874
        *((0.1065*0.5000)*(1.1068*V)))),0),2.5000)),(-1))))))
VTF_sc = (0.1095*VTF) = ((0.0923*20)*(1.1860*(0.0500*(15.6000*pow((1+max((2*(0.6366
        *((0.1050*0.500)*((1.2343*0.500)*(((6.9184*U)+(6.9184*U))*(1.7527*UMOD)))))),0)),(-1))))))
V_sc = (1.1068*V) = integ((11.4602*(0.9892*((0.0985*((0.8959*(-1.0000))*(1.1068*V)))
        +(0.9457*((0.9426*1.0000)*(0.1095*VTF))))))),((7.8379*2)*(0.1412*0)))
U_sc = (6.9184*U) = integ((11.4602*(0.9446*((0.0972*((0.9500*(-1.0000))*(6.9184*U)))
        +(9.6928*((0.5948*1.0000)*(0.1109*UTF))))))),((8.2713*2)*(0.8364*0)))
compV_sc = (0.0600*compV) = (0.6000*emit((0.0959*((0.5655*1.6667)*(1.1068*V)))))
```

Figure 6-62: Scaled signal dynamics for `gentog` benchmark

6.3.10 Genetic Toggle Switch (gentog)

Figure 6-52 presents the original dynamical system for the gentog application. The gentog application defines the VPERT, PERT, U, and V state variables and the IPTG, UMOD,UTFVTF, and compV intermediate variables.

```
1  func utf(T) = 15.62/pow((1+max(T,0)),2.5)
2  func vtf(T) = 15.6/(1+max(T,0))
3  func umod(T) = pow((1+max(T,0)),-2.0015)
4
5  var VPERT = integ(-PERT, 0.0)
6  var PERT = integ(VPERT, 0.5)
7  var IPTG = 0.5+PERT
8  var UMOD = call(umod,IPTG)
9  var UTF = call(utf,V)
10 var VTF = call(vtf,(U*UMOD))
11 var V = integ(VTF - V, 0.0)
12 var U = integ(UTF - U, 0.0)
13 var compV = emit(V)
```

The gentog application uses the utf, vtf, and umod functions which compute the $15.62 \cdot (1+T)^{-2.5}$, $15.6 \cdot (1+T)^{-1}$, and $(1+T)^{-2.0015}$ functions respectively. The umod function is invoked with the IPTG function as an argument, the utf function is invoked with the V function as an argument, and the vtf function is invoked with the U*UMOD term as an argument. The max(T,0) invocations ensure that the input argument is greater than or equal to zero. This is necessary because the U, V, and IPTG variables all capture protein or small molecule levels which cannot take on negative values.

Unscaled ADP

Figures 6-59 and 6-58 present the unscaled ADP for the gentog application. The ADP has a total of 28 blocks and 32 connections. The circuit contains 7 multipliers, 4 integrators, 3 ADCs, 4 DACs, 3 LUTs, 3 current copiers, 1 observation block, and 3 routing (tin and tout) blocks. The compiler uses the tin and tout route blocks are to forward the signal implementing VTF from port z of DAC (0,2,2,0) to port y of multiplier (0,3,2,0). The compiler partitions the circuit across tiles because tiles (0,2) and (0,3) each only have 2 LUTs and 2 ADCs. The compiler must therefore use more than one tile to to use 4 ADCs, 4 DACs, and 3 LUTs. The compiler uses the remaining tout route block to forward the signal implementing V to the observation block. The unscaled ADP uses three copiers to produce 2 copies of the V, PERT, and U signals. The ADP uses Kirchoff's law to implement the addition operators in the relations governing U,V,IPTG, and VTF. The circuit

instantiates 11 constant data fields to provide the values -1.0, 0.0, 0.25, 1.0, and 1.67 to the circuit.

Figure 6-60 presents the dynamics of the signals from the unscaled ADP for the **smmrxn** benchmark. The unscaled ADP implements the PERT, VPERT, IPTG, UMOD, UTF, VTF, U, and V variables as analog currents and the compV variable as an analog voltage:

```
1  VPERT = integ(((-1.00)*PERT),(2*0))
2  PERT = integ(VPERT,(2*0.25))
3  IPTG = ((2*0.25)+PERT)
4  UMOD = (20*(0.05*pow((1+max((2*(0.50*IPTG)),0)),(-2.00))))
5  UTF = (20*(0.05*(15.62*pow((1+pow(max((2*(0.50*V)),0),2.50)),(-1)))))
6  VTF = (20*(0.05*(15.60*pow((1+max((2*(0.50*((0.50*(U+U))*UMOD))),0)),(-1)))))
7  V = integ((((-1.00)*V)+(1.00*VTF)),(2*0))
8  U = integ((((-1.00)*U)+(1.00*UTF)),(2*0))
9  compV = emit((0.60*(1.67*V)))
```

The PERT, VPERT, IPTG, UMOD, UTF, VTF, U, V, and compV signal expressions all fail to syntactically match the original dynamical system relations:

- VPERT: The compiler implements the 0 initial condition with the 2*0.0 expression where 0.0 is a data field value and 2 is a device term introduced by a integrator block. The compiler negates the PERT signal by multiplying it with -1.0. The analog current at port z of integrator (0,2,1,0) implements the VPERT variable.
- PERT: The compiler implements the 0.5 initial condition with the 2*0.25 expression where 0.25 is a data field value and 2.0 is a device term introduced by an integrator. The analog current at port z of integrator (0,2,3,0) implements the PERT variable.
- IPTG: The compiler implements the 0.5 term as 2*0.25 where where 0.25 is a data field value and 2.0 is a device term introduced by a DAC. The compiler implements the addition operation with Kirchhoff's law. The analog current at port x of ADC (0,2,2,0) implements the IPTG variable.
- UMOD: The compiler introduces the 2 and 0.5 terms into the data field expression to compensate for the 0.50 and 20 device terms introduced by the ADC and DAC blocks respectively. The compiler multiplies the expression data field input with the coefficient 2.0 to compensate for the 0.5 device term introduced by the DAC. The compiler multiplies the expression data field output with the coefficient 0.05 to compensate for the device term introduced by the ADC. The analog current at port z of DAC (0,2,2,0) implements the UMOD variable.

- UTF: The compiler introduces the 2 and 0.5 terms into the data field expression to compensate for the 0.50 and 20 device terms introduced by the ADC and DAC blocks respectively. The analog current at port z of DAC (0,3,0,0) implements the UTF variable.

- VTF:The compiler introduces the 2 and 0.5 terms into the data field expression to compensate for the 0.50 and 20 device terms introduced by the ADC and DAC blocks respectively. The compiler also rewrites the U*UMOD argument as 0.5*(U+U)*UMOD. The compiler doubles the U signal by adding it with itself – this cancels out the 0.5 device term introduced by the multiplier. The analog current at port z of DAC (0,3,2,0) implements the VTF variable.

- U: The compiler implements UTF-U by negating the U term and adding it with UTF. The compiler negates U by introducing the -1.0 data field value and multiplying it with U. This is necessary because subtraction is not explicitly supported in the device. The analog current at port z of integrator (0,3,1,0) implements the U variable.

- V: The compiler implements VTF-V by negating the V term and adding it with VTF. The compiler negates V by introducing the -1.0 data field value and multiplying it with V. This is necessary because subtraction is not explicitly supported in the device. The analog current at port z of integrator (0,3,0,0) implements V.

- compV: The compiler introduces the 1.67 data field value to cancel out the 0.60 device term introduced by the observation block. The analog voltage at port z of observation block (0,3,2,0) implements the compV variable.

Scaled ADP

Figure 6-62 presents the scaled ADP for the smmrxn application. The scaled ADP has a total of 90 magnitude scale factors. The scaled ADP has 11 data field scale factors, 9 variable scale factors, and one time scale factor (0.08729). The speed of computation is 0.08729x the baseline integration speed of the device. The compiler also changes the block mode for a subset of ADP blocks. The block mode modifications which change the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 2, 3, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 2, 1, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,h,+)
mult	(0, 2, 1, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 0, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 1, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 1, 0)	[(x,m,m), (x,h,h)]	(x,m,h)
mult	(0, 3, 0, 0)	[(x,m,m), (x,h,h)]	(x,h,m)
dac	(0, 2, 3, 0)	[(const,m)]	(const,h)
dac	(0, 2, 2, 0)	[(dyn,h)]	(dyn,m)
dac	(0, 3, 0, 0)	[(dyn,h)]	(dyn,m)
dac	(0, 3, 2, 0)	[(dyn,h)]	(dyn,m)
adc	(0, 2, 2, 0)	[(m)]	(h)
adc	(0, 3, 0, 0)	[(m)]	(h)
adc	(0, 3, 2, 0)	[(m)]	(h)

The scaled ADP changes 5 multiplier modes, 4 integrator modes, 3 DAC modes, and 3 ADC modes to scale the circuit:

- **integrators** `(0,2,3,0)`, `(0,3,0,0)`, **and** `(0,3,1,0)`: The compiler changes the mode from `(m,m,+)` to `(h,h,+)`. This modification scales the initial condition by `10` and expands the operating range of the input port `x` and the output port `z`.
- **integrator** `(0,2,1,0)`: The compiler changes the mode from `(m,m,+)` to `(h,m,+)`. This modification scales the derivative of the output signal by `0.1` and expands the operating range of the input port `x`.
- **multipliers** `(0,2,1,1)`,`(0,3,0,1)`, `(0,3,1,1)`, **and** `(0,3,0,0)`: The compiler changes the mode from `(x,m,m)` or `(x,h,h)` to `(x,h,m)`. This modification scales the output signal by `0.1` and reduces the operating range of the output port `z` (relative to the mode `(x,h,h)`).
- **multiplier** `(0,3,1,0)`: The compiler changes the mode from `(x,m,m)` or `(x,h,h)` to `(x,m,h)`. This modification scales the output signal by `10.0` and reduces the operating range of the input port `x` (relative to the mode `(x,h,h)`.
- **DAC** `(0,2,3,0)`: The compiler changes the mode from `(const,m)` to `(const,h)`. This modification scales the output signal by `10` an expands the operating range of the output port `z`.

- **DACs** `(0,2,2,0),(0,3,0,0),` **and** `(0,3,2,0)`: The compiler changes the mode from `(dyn,h)` to `(dyn,m)`. This modification scales the output signal by `0.1` an reduces the operating range of the output port `z`.
- **all ADCs**: The compiler changes the mode from `(m)` to `(h)`. This modification scales the output signal by `0.1` and increases the operating range of the input port `x`.

Figure 6-62 presents the scaled dynamics of the scaled ADP for the `smmrxn` benchmark.

```
1  VPERT_sc = (1.6604*VPERT) = integ((11.4602*(0.1012*(0.1000*((0.7909*(-1.0000))*(18.1021*PERT))))),
2           ((0.7738*2)*(2.1458*0)))
3  PERT_sc = (18.1021*PERT) = integ((11.4602*(0.9513*(1.6604*VPERT))),((6.8244*2)*(2.6526*0.2500)))
4  IPTG_sc = (18.1021*IPTG) = (((9.2446*2)*(1.9581*0.2500))+(18.1021*PERT))
5  UMOD_sc = (1.7527*UMOD) = ((0.0959*20)*(18.2724*(0.0500*pow((1+max((2*(0.5305
6           *((0.1041*0.5000)*(18.1021*IPTG)))),0)),(-2.0015))))))
7  UTF_sc = (0.1109*UTF) = ((0.0919*20)*(1.2069*(0.0500*(15.6200*pow((1+pow(max((2*(8.4874
8           *((0.1065*0.5000)*(1.1068*V))))),0),2.5000)),(-1))))))
9  VTF_sc = (0.1095*VTF) = ((0.0923*20)*(1.1860*(0.0500*(15.6000*pow((1+max((2*(0.6366
10          *((0.1050*0.500)*((1.2343*0.500)*(((6.9184*U)+(6.9184*U))*(1.7527*UMOD)))))),0)),(-1))))))
11 V_sc = (1.1068*V) = integ((11.4602*(0.9892*((0.0985*((0.8959*(-1.0000))*(1.1068*V)))
12          +(0.9457*((0.9426*1.0000)*(0.1095*VTF))))))),((7.8379*2)*(0.1412*0)))
13 U_sc = (6.9184*U) = integ((11.4602*(0.9446*((0.0972*((0.9500*(-1.0000))*(6.9184*U)))
14          +(9.6928*((0.5948*1.0000)*(0.1109*UTF))))))),((8.2713*2)*(0.8364*0)))
15 compV_sc = (0.0600*compV) = (0.6000*emit((0.0959*((0.5655*1.6667)*(1.1068*V)))))
```

The compiler scales the `VPERT`, `PERT`, `IPTG`, `UMOD`, `UTF`, `VTF`, `V`, `U`, and `compV` variables by `1.6604`, `18.1021`, `18.1021`, `1.7527`, `0.1109`, `0.1095`, `1.1068`, `6.9184`, and `0.0600` respectively. The compiler reports a time scale factor of `0.08729`. The compiler therefore changes the speed of the computation by a factor of 11.4602^{-1} or `0.08729`. The scaled signal dynamics use eleven data field scale factors, 9 variable scale factors, 6 injected coefficients, and one time scale factor `0.08729`. I summarize the scaled data field values below:

- In the `VPERT` and `PERT` signals, the compiler scales the `-1.0`, `0.0`, `0.25` data field values by `0.7909`, `2.1458`, and `2.6526` respectively.
- In the `IPTG` and `V` signals, the compiler scales the `0.25`, `-1`, `1.0`, and `0` data field values by `1.9581`, `0.8959`, `0.9426`, and `0.1412` respectively.
- In the `U` and `compV` signals, the compiler scales the `-1.0`, `1.0`, `0`, and `1.67` values by `0.9500`, `0.5948`, `0.8364`, and `0.5655` respectively.
- For the `UMOD` signal, the compiler injects the `0.5305` and `18.2724` coefficients into the data field expression. The `0.5305` coefficient eliminates the scaling transform and compensation terms from the input expression and the `18.2724` term scales the data field expression result by 18.2724x.

- For the UTF signal, the compiler injects the 8.4874 and 1.2069 coefficients into the data field expression. The 8.4874 coefficient eliminates the scaling transform and compensation terms from the input expression and the 1.2069 term scales the data field expression result by 1.2069x.
- For the VTF signal, the compiler injects the 0.6366 and 1.1860 coefficients into the data field expression. The 0.6366 coefficient eliminates the scaling transform and compensation terms from the input expression and the 1.1860 term scales the data field expression result by 1.1860x.

The compensation terms with values between 0.7738-1.2343 capture only the behavioral variations present in the device. All other compensation terms capture both the effect of changing the block mode and model the behavioral deviations found on the device. The compensation terms between 0.0923-0.1041 capture mode changes which introduce the 0.1 coefficient into the block input-output relations. The compensation terms between 6.8244-9.244 compensate for the mode changes introduce the 10 coefficient into the input-output relations.

Preservation:The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- VPERT **variable (III)**: The scale expression for both the initial condition and the derivative of VPERT all simplify to 1.6604, the magnitude scale factor of the VPERT variable:

$$\begin{aligned} \text{VPERT'} \quad 1.6604 &= 11.4602*0.1012*0.1000*0.7909*18.1021 \\ \text{VPERT(0)} \quad 1.6604 &= 0.7738*2.1458 \end{aligned}$$

- PERT **variable (III)**: The scale expression for both the initial condition and the derivative of PERT all simplify to 18.1021, the magnitude scale factor of the PERT variable:

$$\begin{aligned} \text{PERT'} \quad 18.1021 &= 11.4602*0.9513*1.6604 \\ \text{PERT(0)} \quad 18.1021 &= 6.8244*2.6526 \end{aligned}$$

- IPTG **variable (II)**: The scale expression for each of the terms in the signal expression for IPTG all simplify to 18.1021, the magnitude scale factor of the IPTG variable:

$$\begin{aligned} 0.5 \quad 18.1021 &= 9.2446*1.9581 \\ \text{PERT} \quad 18.1021 &= 18.1021 \end{aligned}$$

- UMOD **variable (I)**:The scale expression for the signal simplifies to 1.7527, the magnitude scale factor for the UMOD variable:

```
UMOD   1.7527   = 0.0959*18.2724
```

The body of the data field expression can be ignored when proving preservation because the scaling transform is eliminated at the data expression input:

```
1.00   = 0.5305*0.1041*18.1021
```

The injected value 0.5305 cancels out both the scaling transform and the compensation terms introduced by the ADC.

- UTF **variable (I)**:The scale expression for the signal simplifies to 0.1109, the magnitude scale factor for the UTF variable:

```
UTF   0.1109   = 0.0919*1.2069
```

The body of the data field expression can be ignored when proving preservation because the scaling transform is eliminated at the data expression input:

```
1.00   = 8.4874*0.1065*1*1.1068
```

The injected value 8.4874 cancels out both the scaling transform and the compensation terms introduced by the ADC.

- VTF **variable (I)**:The scale expression for the signal simplifies to 0.1095, the magnitude scale factor for the VTF variable:

```
VTF   0.1095   = 0.0923*1.1860
```

The body of the data field expression can be ignored when proving preservation because the scaling transform is eliminated at the data expression input:

```
1.00   = 0.6366*0.1050*1.2343*1*1*6.9184*1*1*1.7527
```

The injected value 0.6366 cancels out both the scaling transform and the compensation terms introduced by the ADC, multiplier, and current copiers.

- V **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to 1.1068, the magnitude scale factor of the V variable:

```
 -V    1.1068   = 11.4602*0.9892*0.0985*0.8959*1.1068
 VTF   1.1068   = 11.4602*0.9892*0.9457*0.9426*0.1095
 V(0)  1.1068   = 7.8379*0.1412
```

- `U` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to `6.9184`, the magnitude scale factor of the `U` variable:

```
 -U    6.9184   = 11.4602*0.9446*0.0972*0.9500*6.9184
 UTF   6.9184   = 11.4602*0.9446*9.6928*0.5948*0.1109
 U(0)  6.9184   = 8.2713*0.8364
```

- `compV` **variable (I)**:The scale expression for the signal simplifies to `0.0600`, the magnitude scale factor for the `compV` variable:

```
 compV  0.0600   = 0.0959*0.5655*1.1068
```

```
    var bulkB = integ(-0.015*bulkB, 1.0)
    var freeB = integ(0.15*bulkB-0.058*freeB, 1.0)
    var bndB = integ(0.058*freeB-0.141*bndB, 0.0)
    var transB = integ(0.141*bndB-0.013*transB, 0.0)
    var lyticB = integ(0.013*transB, 0.0)
    var MTRANSB = emit(transB)

    interval bulkB = [-1.0,1.0]
    interval freeB = [-0.02,0.02]
    interval bndB = [-0.03,0.03]
    interval transB = [-0.06,0.06]
    interval lyticB = [-0.004,0.004]

    time 20
```

Figure 6-63: Dynamical system for `bont4` benchmark

```
conn block mult port z loc (0, 3, 3, 0) with block integ port x loc (0, 3, 2, 0);
conn block mult port z loc (0, 3, 0, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 2, 0) with block integ port x loc (0, 3, 3, 0);
conn block mult port z loc (0, 3, 3, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 1, 1) with block integ port x loc (0, 3, 0, 0);
conn block mult port z loc (0, 3, 0, 1) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 3, 1, 0) with block integ port x loc (0, 3, 1, 0);
conn block mult port z loc (0, 2, 2, 0) with block integ port x loc (0, 2, 3, 0);
conn block mult port z loc (0, 3, 2, 1) with block tout port x loc (0, 3, 0, 0);
conn block tout port z loc (0, 3, 0, 0) with block extout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 2, 0) with block fanout port x loc (0, 3, 2, 0);
conn block integ port z loc (0, 3, 3, 0) with block fanout port x loc (0, 3, 0, 1);
conn block integ port z loc (0, 3, 0, 0) with block fanout port x loc (0, 3, 0, 0);
conn block integ port z loc (0, 3, 1, 0) with block fanout port x loc (0, 3, 1, 0);
conn block fanout port z0 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 3, 0);
conn block fanout port z1 loc (0, 3, 2, 0) with block mult port x loc (0, 3, 0, 0);
conn block fanout port z0 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 2, 0);
conn block fanout port z1 loc (0, 3, 0, 1) with block mult port x loc (0, 3, 1, 1);
conn block fanout port z0 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 3, 1);
conn block fanout port z1 loc (0, 3, 0, 0) with block mult port x loc (0, 3, 0, 1);
conn block fanout port z0 loc (0, 3, 1, 0) with block mult port x loc (0, 3, 1, 0);
conn block fanout port z1 loc (0, 3, 1, 0) with block tout port x loc (0, 3, 0, 1);
conn block tout port z loc (0, 3, 0, 1) with block tin port x loc (0, 2, 0, 0);
conn block tin port z loc (0, 2, 0, 0) with block mult port x loc (0, 2, 2, 0);
conn block fanout port z2 loc (0, 3, 1, 0) with block mult port x loc (0, 3, 2, 1);
```

Figure 6-64: Connections from unscaled/scaled ADP for `bont4` benchmark

```
config block integ @ (0, 3, 2, 0) {
  modes [(m,m,+)]; source bulkB at z; set z0 at 0.500; }
config block integ @ (0, 3, 3, 0) {
  modes [(m,m,+)]; source freeB at z; set z0 at 0.000; }
config block integ @ (0, 3, 0, 0) {
  modes [(m,m,+)]; source bndB at z; set z0 at 0.000; }
config block integ @ (0, 3, 1, 0) {
  modes [(m,m,+)]; source transB at z; set z0 at 0.000; }
config block integ @ (0, 2, 3, 0) {
  modes [(m,m,+)]; source lyticB at z; set z0 at 0.000; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.015; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 0.015; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.058; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,m,m), (x,h,h)]; set c at -0.141; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 0.058; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 0.141; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m), (x,h,h)]; set c at -0.013; }
config block mult @ (0, 2, 2, 0) {
  modes [(x,m,m), (x,h,h)]; set c at 0.013; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m), (x,h,h)]; set c at 1.667; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; source MTRANSB at z; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source bulkB at z0; source bulkB at z1;
  source bulkB at z2; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m), (+,+,+,h)]; source freeB at z0; source freeB at z1;
  source freeB at z2; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source bndB at z0; source bndB at z1;
  source bndB at z2; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,+,m), (+,+,+,h)]; source transB at z0; source transB at z1;
  source transB at z2; }
config block tout @ (0, 3, 0, 0) {
  modes [(*)]; }
config block tout @ (0, 3, 0, 1) {
  modes [(*)]; }
config block tin @ (0, 2, 0, 0) {
  modes [(*)]; }
```

Figure 6-65: Block Configurations from unscaled ADP for `bont4` benchmark. Refer to Figure 6-64 for scaled ADP connections.

```
bulkB = integ(((-0.01)*bulkB),(2*0.50))
freeB = integ(((0.01*bulkB)+((-0.06)*freeB)),(2*0))
bndB = integ((((-0.14)*bndB)+(0.06*freeB)),(2*0))
transB = integ(((0.14*bndB)+((-0.013)*transB)),(2*0))
lyticB = integ((0.013*transB),(2*0))
MTRANSB = emit((0.60*(1.67*transB)))
```

Figure 6-66: Signal dynamics of the unscaled ADP for `bont4` benchmark

```
config block integ @ (0, 3, 2, 0) {
  modes [(h,m,+)]; scale x = 8.234; source bulkB at z; scale z = 1.361;
  set z0 at 0.500; scale z0 = 1.885; }
config block integ @ (0, 3, 3, 0) {
  modes [(h,m,+)]; scale x = 61.806; source freeB at z; scale z = 9.500;
  set z0 at 0.000; scale z0 = 15.600; }
config block integ @ (0, 3, 0, 0) {
  modes [(h,h,+)]; scale x = 163.596; source bndB at z; scale z = 262.566;
  set z0 at 0.000; scale z0 = 33.500; }
config block integ @ (0, 3, 1, 0) {
  modes [(h,m,+)]; scale x = 172.820; source transB at z; scale z = 26.492;
  set z0 at 0.000; scale z0 = 44.733; }
config block integ @ (0, 2, 3, 0) {
  modes [(h,m,+)]; scale x = 1934.663; source lyticB at z; scale z = 313.918;
  set z0 at 0.000; scale z0 = 386.101; }
config block mult @ (0, 3, 3, 0) {
  modes [(x,m,m)]; scale x = 1.361; scale y = 1.000; scale z = 8.234; set c at -0.015;
  scale c = 6.170; }
config block mult @ (0, 3, 0, 0) {
  modes [(x,h,h)]; scale x = 1.361; scale y = 1.000; scale z = 61.806; set c at 0.015;
  scale c = 62.839; }
config block mult @ (0, 3, 2, 0) {
  modes [(x,m,m)]; scale x = 9.500; scale y = 1.000; scale z = 61.806; set c at -0.058;
  scale c = 6.691; }
config block mult @ (0, 3, 3, 1) {
  modes [(x,h,m)]; scale x = 262.566; scale y = 1.000; scale z = 163.596; set c at -0.141;
  scale c = 6.343; }
config block mult @ (0, 3, 1, 1) {
  modes [(x,h,h)]; scale x = 9.500; scale y = 1.000; scale z = 163.596; set c at 0.058;
  scale c = 16.251; }
config block mult @ (0, 3, 0, 1) {
  modes [(x,h,m)]; scale x = 262.566; scale y = 1.000; scale z = 172.820; set c at 0.141;
  scale c = 6.685; }
config block mult @ (0, 3, 1, 0) {
  modes [(x,m,m)]; scale x = 26.492; scale y = 1.000; scale z = 172.820; set c at -0.013;
  scale c = 6.638; }
config block mult @ (0, 2, 2, 0) {
  modes [(x,h,h)]; scale x = 26.492; scale y = 1.000; scale z = 1934.663; set c at 0.013;
  scale c = 72.506; }
config block mult @ (0, 3, 2, 1) {
  modes [(x,m,m)]; scale x = 26.492; scale y = 1.000; scale z = 14.169; set c at 1.667;
  scale c = 0.566; }
config block extout @ (0, 3, 2, 0) {
  modes [(*)]; scale x = 14.169; source MTRANSB at z; scale z = 14.169; }
config block fanout @ (0, 3, 2, 0) {
  modes [(+,+,+,m)]; scale x = 1.361; source bulkB at z0; scale z0 = 1.361;
  source bulkB at z1; scale z1 = 1.361; source bulkB at z2; scale z2 = 1.361; }
config block fanout @ (0, 3, 0, 1) {
  modes [(+,+,+,m)]; scale x = 9.500; source freeB at z0; scale z0 = 9.500;
  source freeB at z1; scale z1 = 9.500; source freeB at z2; scale z2 = 9.500; }
config block fanout @ (0, 3, 0, 0) {
  modes [(+,+,+,h)]; scale x = 262.566; source bndB at z0; scale z0 = 262.566;
  source bndB at z1; scale z1 = 262.566; source bndB at z2; scale z2 = 262.566; }
config block fanout @ (0, 3, 1, 0) {
  modes [(+,+,+,m)]; scale x = 26.492; source transB at z0; scale z0 = 26.492;
  source transB at z1; scale z1 = 26.492; source transB at z2; scale z2 = 26.492; }
config block tout @ (0, 3, 0, 0) { modes [(*)]; scale x = 14.169; scale z = 14.169; }
config block tout @ (0, 3, 0, 1) { modes [(*)]; scale x = 26.492; scale z = 26.492; }
config block tin @ (0, 2, 0, 0) { modes [(*)]; scale x = 26.492; scale z = 26.492; }
timescale 0.616322
```

Figure 6-67: Block configurations and `timescale` statement from scaled ADP of `bont4` benchmark. Refer to Figure 6-64 for scaled ADP connections.

```
1  bulkB_sc = (1.3609*bulkB) = integ((1.6225*(0.1019*(0.9806*((6.1698*(-0.0150))*(1.3609*bulkB))))),
2           ((0.7219*2)*(1.8852*0.5000)))
3  freeB_sc = (9.5000*freeB) = integ((1.6225*(0.0947*((0.7227*((62.8385*0.0150)*(1.3609*bulkB))))
4           +((0.9723*((6.6913*(-0.0580))*(9.5000*freeB)))))),((0.6090*2)*(15.6002*0)))
5  bndB_sc = (262.5659*bndB) = integ((1.6225*(0.9892*((0.0982*((6.3428*(-0.1410))*(262.5659*bndB))))
6           +((1.0596*((16.2513*0.0580)*(9.5000*freeB)))))),((7.8379*2)*(33.4996*0)))
7  transB_sc = (26.4925*transB) = integ((1.6225*(0.0945*((0.0985*((6.6850*0.1410)*(262.5659*bndB))))
8           +((0.9827*((6.6381*(-0.0130))*(26.4925*transB)))))),((0.5922*2)*(44.7327*0)))
9  lyticB_sc = (313.9181*lyticB) = integ((1.6225*(0.1000*(1.0072*((72.5060*0.0130)*(26.4925*transB))))),
10          ((0.8130*2)*(386.1007*0)))
11 MTRANSB_sc = (14.1694*MTRANSB) = (0.6000*emit((0.9457*((0.5655*1.6667)*(26.4925*transB)))))
```

Figure 6-68: Scaled signal dynamics for `bont4` benchmark

6.3.11 Botulism Neurotoxin (bont4)

Figure 6-63 presents the original dynamical system for the bont4 application. The bont4 application defines the bulkB, freeB, bndB, transB, lyticB state variables and the MTRANSB intermediate variable:

```
1 var bulkB = integ(-0.015*bulkB, 1.0)
2 var freeB = integ(0.015*bulkB-0.058*freeB, 1.0)
3 var bndB = integ(0.058*freeB-0.141*bndB, 0.0)
4 var transB = integ(0.141*bndB-0.013*transB, 0.0)
5 var lyticB = integ(0.013*transB, 0.0)
6 var MTRANSB = emit(transB)
```

Unscaled ADP

Figure 6-65 presents the unscaled ADP for the bont4 application. The ADP has a total of 22 blocks and 25 connections. The circuit contains 9 multipliers, 5 integrators, 4 current copiers, 1 observation block, and 3 route (tin andtout) blocks. The compiler uses the tin and tout route blocks to forward the signal implementing transB from the current copier at (0,3,1,0) to the multiplier at (0,2,2,0). The forwarded signal is then used to compute lyticB. The compiler splits the circuit across multiple tiles the circuit uses 9 multipliers but each tile contains at most 8 multipliers. The compiler uses the remaining tout block to forward the signal implementing transB to the observation block. The unscaled ADP uses the three copiers to produce 2 copies of the bulkB, freeB, and bndB signals and one copier to produce 3 copies of the transB signal. The compiler uses Kirchhoff's law to implement the addition operators in the relations governing freeB, bndB, and transB. The circuit instantiates 14 constant data fields to provide the -0.015, 0.50, 0.015, -0.058, 0.0, 0.058, 0.141, -0.013, 0.013, and 1.67 values to the circuit.

Figure 6-66 presents the dynamics of the signals from the unscaled ADP for the bont4 benchmark. The compiler implements the bulkB, freeB, bndB, transB, and lyticB variables as analog currents and the MTRANSB variable as an analog voltage:

```
1 bulkB = integ(((-0.01)*bulkB),(2*0.50))
2 freeB = integ(((0.01*bulkB)+((-0.06)*freeB)),(2*0))
3 bndB = integ((((-0.14)*bndB)+(0.06*freeB)),(2*0))
4 transB = integ(((0.14*bndB)+((-0.013)*transB)),(2*0))
5 lyticB = integ((0.013*transB),(2*0))
6 MTRANSB = emit((0.60*(1.67*transB)))
```

The unscaled ADP has 6 labelled analog currents which implement the bulkB, freeB, bndB,

transB, lyticB, and MTRANSB dynamical system variables. Note that the bulkB, freeB, bndB, transB, and MTRANS signal expressions do not syntactically match the original dynamical system relations:

- bulkB: The compiler implements the 1.0 initial condition as 2*0.5 where 0.5 is a data field value and 2.0 is a device term introduced by the integrator block. The analog current at port z of integrator (0,3,2,0) implements the bulkB variable.
- freeB: The compiler implements a negated version of the 0.058*freeB term as (-0.058)*freeB. The compiler then adds negated term (-0.058)*freeB to the 0.015*bulkB term with Kirchhoff's law to implement the derivative expression. This is necessary because subtraction is not explicitly supported in the device. The compiler implements initial condition 0.0 as 2.0*0.0 where 0.0 is a data field value and 2.0 is a device term introduced by the integrator. The analog current at port z of integrator (0,3,3,0) implements the freeB variable.
- bndB: The compiler implements a negated version of the 0.141*bndB term as (-0.141)*bndB. The compiler then adds negated term (-0.141)*bndB to the 0.058*freeB term with Kirchhoff's law to implement the derivative expression. This is necessary because subtraction is not explicitly supported in the device. The analog current at port z of integrator (0,3,0,0) implements the bndB variable.
- transB: The compiler implements a negated version of the 0.013*transB term as (-0.013)*transB. The compiler then adds negated term (-0.013)*transB to the 0.141*bndB term with Kirchhoff's law to implement the derivative expression. This is necessary because subtraction is not explicitly supported in the device. The analog current at port z of integrator (0,3,1,0) implements the transB variable.
- lyticB: The compiler implements initial condition 0.0 as 2.0*0.0 where 0.0 is a data field value and 2.0 is a device term introduced by the integrator. The analog current at port z of integrator (0,2,3,0) implements the lyticB variable.
- MTRANSB: The compiler introduces the 1.67 data field value to compensate for the 0.60 device term introduced by the observation block. The analog voltage at port z of observation block (0,3,2,0) implements the MTRANSB variable.

Scaled ADP

Figure 6-64 and Figure 6-67 scaled ADP for the bont4 benchmark. The scaled ADP has a total of 75 magnitude scale factors. The scaled ADP has 14 data field scale factors, 6 variable scale factors, and one time scale factor (0.616322). The speed of the scaled computation is therefore 0.616322x the baseline integration speed of the device. The compiler also changes the block modes for a subset

of the ADP blocks to better scale te circuit. The block mode modifications which change the block input-output relations are summarized below:

block	location	mode (unscaled ADP)	mode (scaled ADP)
integ	(0, 3, 2, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 3, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 3, 0, 0)	[(m,m,+)]	(h,h,+)
integ	(0, 3, 1, 0)	[(m,m,+)]	(h,m,+)
integ	(0, 2, 3, 0)	[(m,m,+)]	(h,m,+)
mult	(0, 3, 3, 1)	[(x,m,m), (x,h,h)]	(x,h,m)
mult	(0, 3, 0, 1)	[(x,m,m), (x,h,h)]	(x,h,m)

- **integrators** (0,3,2,0), (0,3,3,0), (0,3,1,0), **and** (0,2,3,0): The compiler changes the integrator mode from (m,m,+) to (h,m,+). This modification scales the derivative of the output signal by 0.1 and expands the operating range of input port x.
- **integrator** (0,3,0,0): The compiler changes the integrator mode from (m,m,+) to (h,h,+). This modification scales the initial condition by 10.0 and expands the operating range of the input port x and the output port z.
- **all multipliers**: The compiler changes the multiplier mode from (x,m,m) or (x,h,h) to (x,h,m). This modification scales the output signal by 0.1 but reduces the operating range of the output port z (relative to the mode (x,h,h)).

Figure 6-68 presents the scaled dynamics of the scaled ADP for the bont4 benchmark.

```
1  bulkB_sc = (1.3609*bulkB) = integ((1.6225*(0.1019*(0.9806*((6.1698*(-0.0150))*(1.3609*bulkB))))),
2          ((0.7219*2)*(1.8852*0.5000)))
3  freeB_sc = (9.5000*freeB) = integ((1.6225*(0.0947*((0.7227*((62.8385*0.0150)*(1.3609*bulkB)))
4          +(0.9723*((6.6913*(-0.0580))*(9.5000*freeB))))))),((0.6090*2)*(15.6002*0)))
5  bndB_sc = (262.5659*bndB) = integ((1.6225*(0.9892*((0.0982*((6.3428*(-0.1410))*(262.5659*bndB)))
6          +(1.0596*((16.2513*0.0580)*(9.5000*freeB))))))),((7.8379*2)*(33.4996*0)))
7  transB_sc = (26.4925*transB) = integ((1.6225*(0.0945*((0.0985*((6.6850*0.1410)*(262.5659*bndB)))
8          +(0.9827*((6.6381*(-0.0130))*(26.4925*transB))))))),((0.5922*2)*(44.7327*0)))
9  lyticB_sc = (313.9181*lyticB) = integ((1.6225*(0.1000*(1.0072*((72.5060*0.0130)*(26.4925*transB))))),
10         ((0.8130*2)*(386.1007*0)))
11 MTRANSB_sc = (14.1694*MTRANSB) = (0.6000*emit((0.9457*((0.5655*1.6667)*(26.4925*transB)))))
```

The compiler scales the bulkB, freeB, bndB, transB, lyticB, and MTRANSB variables by 1.3609, 9.500, 262.5659, 26.4925, 313.9181, and 14.1694 respectively. The compiler reports a time scale factor of 0.616322. The compiler therefore scales the speed of the computation by a factor of 1.6225^{-1} or 0.616322. The scaled ADP has 14 data field scale factors, 6 variable scale factors, and one time scale factor 0.616322. I summarize the scaled data field values below:

- In the bulkB and freeB signals, the compiler scales the -0.0150, 0.5000, 0.0150, -0.580, and 0.0 data field values by 6.1698, 1.8852, 62.8385, 6.6913, and 15.6002 respectively.
- In the bndB signal, the compiler scales the -0.1410, 0.0580, and 0 data field values by 6.3428,16.2513, and 33.4996 respectively.
- In the transB signal, the compiler scales the 0.1410, -0.0130, and 0.0 values by 6.6850,6.6381, and 44.7327 respectively.
- In the lyticB signal, the compiler scales the 0.0130 and 0 data field value by 72.5060 and 386.1007 respectively.
- The compiler scales the 1.667 data field value in the MTRANSB by 0.5655.

The compensation terms with values between 0.6090-1.0596 capture only the behavioral variations present in the device. All other compensation terms capture both the effect of changing the block mode and model the behavioral deviations found on the device. The compensation terms with values between 0.0947-0.1000 compensate for mode changes which scale signals by 0.1. The compensation term 7.839 compensates for the mode change which scales signals by 10.

Preservation:The scale factors (red) and compensation terms (grey) can be factored out of the right- and left-hand side of each relation and eliminated from both sides of the equation:

- bulkB **variable (III)**: The scale expression for both the initial condition and the derivative of bulkB all simplify to 1.3609, the magnitude scale factor of the bulkB variable:

```
 bulkB'   1.3609  = 1.6225*0.1019*0.9806*6.16981.3609
bulkB(0)  1.3609  = 0.7219*1.8852
```

- freeB **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to 9.5000, the magnitude scale factor of the freeB variable:

```
0.015*bulkB   9.5000  = 1.6225*0.0947*0.7227*62.8385*1.3609
-0.058*freeB  9.5000  = 1.6225*0.0947*0.9723*6.6913*9.5000
  freeB(0)    9.5000  = 0.6090*15.6002
```

- bndB **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to 262.5659, the magnitude scale factor of the bndB variable:

```
0.058*freeB   262.5659  = 1.6225*0.9892*1.0596*16.2513*9.5000
-0.141*bndB   262.5659  = (1.6225*0.9892*0.0982*6.3428*262.5659
  bndB(0)     262.5659  = 7.8379*33.4996
```

- `transB` **variable (IV)**: The scale expression for each of the derivative terms and the initial condition all simplify to `26.4925`, the magnitude scale factor of the `transB` variable:

```
 0.141*bndB    26.4925  = 1.6225*0.0945*0.0985*6.6850*262.5659
-0.013*transB  26.4925  = 1.6225*0.0945*0.9827*6.6381*-0.0130*26.4925
  transB(0)    26.4925  = 0.5922*44.7327
```

- `lyticB` **variable (III)**: The scale expression for both the initial condition and the derivative of `lyticB` all simplify to `313.9181`, the magnitude scale factor of the `lyticB` variable:

```
  lyticB'    313.9181  = 1.6225*0.1000*1.0072*72.5060*26.4925
 lyticB(0)   313.9181  = 0.8130*386.1007
```

- `MTRANSB` **variable (I)**:The scale expression for the signal simplifies to `14.1694`, the magnitude scale factor for the `MTRANSB` variable:

```
MTRANSB   14.1694  = 0.9457*0.5655*26.4925
```

6.4 General Trends

I next present an overview of the general trends observed in the unscaled and scaled ADPs presented in this chapter.

6.4.1 Unscaled ADPs

The compiler produces unscaled ADPs that directly implement the target dynamical system. In this analysis, the symbolic expressions governing the signal physics rarely syntactically match the signal dynamics in the studied benchmarks. This phenomenon occurs because the HCDCv2 blocks often introduce constant coefficients that cannot be directly set through programming the block. I discuss several common mitigation strategies utilized by the compiler below:

- **Materializing Data Field Values**: The compiler introduces additional blocks into the ADP to cancel out the device terms. These blocks are typically multipliers which have been configured to scale an input signal by a digitally programmable constant data field (`(x,m,m)`, `(x,m,h)`, `(x,h,h)` or `(x,h,m)` modes).

- **Modifying Data Field Values**: The compiler modifies constant fields to cancel out device terms. While this compensation technique does not introduce any additional logic into the circuit, it requires that a constant data field already exists in the right place.

- **Doubling Signals with Copiers**: The compiler sometimes compensates for device terms of the form $(0.5)^n$ by adding a target signal together multiple times. For example, the compiler might implement `X` as `0.5*(X + X)`, where the `X+X` term cancels out the `0.5` device coefficient. This compensation approach often does not require additional logic if the target signal is already copied with a current copier. The compiler implements the addition operation by strategically connecting the copied signals together and leveraging Kirchhoff's law to perform summation.

- **Injecting Values into Data Field Expressions**: The compiler sometimes compensates for device terms by modifying the expressions assigned to expression data fields. This compensation approach requires no additional logic but only can be used if the device term is multiplied with an expression data field term. For example, the expression data field `ln(x)` may be modified to `0.2*5*ln(x)`, where the `5` term cancels out the `0.2` device coefficient.

Note that in some instances, the compiler directly uses the device terms to implement dynamical system dynamics. The compiler's ability to intelligently leverage device terms reduces the number of constant data fields and blocks needed in the produced ADP.

The compiler also rewrites terms to implement dynamics that the HCDCv2 does not directly support. The HCDCv2 does not directly offer a subtraction, so the compiler often rewrites terms to implement subtraction with the logic blocks and physical laws:

- **Addition with Signal Negation**: The compiler often implements subtraction by negating one term with a current copier and then adding it with the other term by leveraging Kirchhoff's law. For example, the compiler would implement an expression `A-B` as `A+((-1)*B)` where `-1` is a device term. This approach does not require additional logic if the negated signal is already supplied to a current copier.

- **Materializing Negative Coefficients**: The compiler will sometimes implement subtraction by introducing additional blocks into the ADP which provide negating coefficients. For example, the compiler would implement and expression `A-B` as `A+((-1)*B)`, where `-1` is a value supplied by a constant data field.

- **Relocating Negation into Data Field Values**: The compiler implements subtraction by negating constant data fields which already exist in the ADP. For example, the compiler would implement `A-0.3B` as `A+(-0.3)*B`, where `-0.3` is the negated constant data field value. This approach does not require any additional logic, but only works if there already is a constant coefficient in the term.

6.4.2 Scaled ADPs

For all applications, the compiler both alters the modes assigned to ADP blocks and scales the signals and data fields such that the resulting simulation executes within the physical constraints imposed by the HCDCv2. Many of the produced scaling transforms are highly diverse and specify many distinct scaling factors. These scaling transforms are sophisticated and compensate for behavioral deviations and changes to the input-output relation introduced by mode changes.

The compiler guarantees that all the computed scaled ADPs preserve the original dynamics from the unscaled ADP. I can confirm this preservation property holds for all of the benchmark applications.

Magnitude Scale Factors

All of the magnitude scale factors that appear in a signal's scaled dynamics are multiplied with a data field value or a port. The magnitude scale factors multiplied with data field values directly change the value written to the data field. The magnitude scale factors multiplied with block ports variables are indirectly set by data fields elsewhere in the circuit and change the signal's dynamic range at

the associated port. The compiler strategically sets the magnitude scale factors to accomplish the following:

- The compiler scales down signals and data fields to fit within the data field operating ranges. For these signals and data fields, the associated magnitude scale factor is typically less than one.
- The compiler scales up signals and data fields to reduce the effect of noise and quantization error. For these signals and data fields, the associated magnitude scale factor is typically more than one.
- The compiler scales signals and data fields to counteract the effect of process variation-induced behavioral deviations and mode changes on the computation.
- The compiler scales signals to manipulate the speed of the computation. The compiler scales the computation speed so that the computation abides by the frequency limitations imposed by the blocks in the circuit.

Modifications to the Block Mode

The compiler often modifies the block modes of the multipliers, integrators, ADCs, and DACs in the circuit to more freely scale the computation. These modifications simultaneously modify the block ports' operating ranges and the input-output relations implemented by the block.

Note that it isn't always advantageous to increase the operating range at all ports. Increasing the operating range of a port also increases the noise floor associated with the port. This change to the operating range can adversely affect signals which under-utilize the expanded operating range. The compiler, therefore, selects modes that offer larger operating ranges only when necessary when scaling the circuit. I discuss the implications of several frequently seen mode modifications below:

- **integrator** `(m,m,+)` **to** `(h,m,+)`: This modification increases the supported operating range of the derivative signal accepted at port `x` and scales the derivative signal by a factor of `0.1`. The compiler makes this modification to slow down the computation and support derivative signals with larger dynamic ranges.
- **integrator** `(m,m,+)` **to** `(h,h,+)`: This modification increases the operating ranges for the derivative and state variable signals at port `x` and `z`. This modification also scales up the initial condition by a factor of `10`. The compiler makes this modification to support derivative and state variable signals with larger dynamic ranges.
- **integrator** `(m,m,+)` **to** `(m,h,+)`: This modification increases the operating range of the output signal at `z` and scales the derivative and initial condition by `10x`. The compiler makes

this modification to increase the dynamic range of a state variable. This modification is not typically used to speed up the computation in practice because many blocks impose strict frequency limitations that keep the computation speed below the baseline integration speed.

- **multiplier** `(x,m,m)/(x,h,h)` **to** `(x,h,m)`: This modification decreases the operating range of the output signal (relative to `(x,h,h)` and scales the output signal by `0.1`. The compiler often applies this modification when it aims to multiply a signal by a small value. Recall the constant data field `c` has limited resolution. This modification enables the compiler to specify the coefficient at finer granularity with data field `c`.
- **multiplier** `(x,m,m)/(x,h,h)` **to** `(x,m,h)`: This modification decreases the operating range of the input signal (relative to `(x,h,h)` and scales the output signal by `10.0`. The compiler applies this modification to scale up the output signal by a factor greater than one.
- **multiplier** `(m,m,m),(m,h,h),(h,m,h)` **to** `(m,m,h)`: This modification reduces the operating range of the input signal (relative to `(h,m,h)` and scales the output signal by `10.0`. The compiler applies this modification to scale up the output signal by a factor greater than one.
- **multiplier** `(m,m,m),(m,h,h),(h,m,h)` **to** `(h,m,m),(m,h,m)`: This modification reduces the operating range of the output signal (relative to `(h,m,h),(m,h,h)` and scales the output signal by `0.1`. The compiler applies this modification to scale down the output signal by a factor greater than one and to reduce the operating ranges of the ports processing the input signals.
- **DAC** `(dyn|const,h)` **to** `(dyn|const,m)`: This modification decreases the operating range of the analog output and scales the analog signal by `0.1`. The compiler applies this modification to reduce the effect of noise on the output signal for output signals which under-utilize the operating range of the DAC.
- **DAC** `(dyn|const,m)` **to** `(dyn|const,h)`: This modification increases the operating range of the analog output and scales the analog signal by `10.0`. The compiler applies this modification to increase the dynamic range of the output signal.
- **ADC** `(m)` **to** `(h)`: This modification increases the acceptable operating range for the input analog signal and scales down the analog signal by `0.1`. The compiler applies this modification to convert signals which have a large dynamic range.

Injected Coefficients

The compiler often injects constant coefficients into the expressions assigned to expression data fields when scaling the circuit. For an expression data field `f(y)`, where the function `f` is programmable and accepts one input `y`, the compiler typically injects coefficients in the following way:

```
x*f(x'*y)
```

The compiler uses the `x` coefficient to scale the result computed by the data field and the `x'` coefficient to eliminate the scaling transform and compensation terms associated with `y`. This expression data field modification eliminates the need for the compiler to propagate the scaling transform through the often highly nonlinear data field expression dynamics. For each of the expression data fields, I can validate that the `x'` successfully cancels out the scaling factors and compensation terms associated with the expression at `y`.

Changing the Computation Speed

The compiler indirectly sets the computation speed by scaling the derivative of each of the state variables by the time scale factor τ. For signals implementing differential equations, the time scale factor appears in the scaled signal dynamics in the following form:

$$x*V = \text{integ}(\tau^{-1}*E,E')$$

The τ^{-1} term controls the ratio of the scaled derivative `V'` to the scaled state variable. Because the original dynamics are preserved in the scaled circuit, the scale expression for the derivative is guaranteed to simplify to `x`. So, if the scale factor factored out of `E` is `x'`, the following equality relation must hold:

$$x = \tau^{-1}*x'$$

In the above relation, increasing the time scale factor increases the magnitude of the derivative relative to the variable magnitude – this accelerates the computation. Decreasing the time scale factor decreases the magnitude of the derivative relative to the variable magnitude – this decelerates the computation. The time scale factor is indirectly set by carefully scaling the data field values that control the magnitude of the derivative and the initial condition of each differential equation.

6.5 Conclusion

The compiler accepts as input a dynamical system and analog device specification. The compiler first produces an unscaled ADP implementation of the provided dynamical system. The physics of the unscaled ADP exactly match the dynamics of the dynamical system provided all the blocks behave ideally under all operating conditions. The unscaled ADP cannot typically be run on the

analog hardware because it doesn't consider the operating range and frequency limitations, noise present in the hardware.

The compiler scales signals and values in the unscaled ADP to produce a scaled ADP. The scaled ADP respects all of the physical limitations and behaviors present in the analog device and preserves the original dynamics of the unscaled ADP (and the dynamical system). The original dynamics of any signal can be recovered from the scaled ADP at runtime by applying a compiler-derived inverting transform. The compiler selectively reprograms blocks in the unscaled ADP to better scale the circuit. The compiler also incorporates the delta model information for the device on hand to scale the circuit. The scaled ADP respects all of the physical constraints of the hardware and can therefore be directly run on the analog device.

In this chapter, I provide a detailed overview of the unscaled and scaled ADPs for the cosine benchmark application introduced in Section 4.1. I then discuss the unscaled and scaled ADPs for each of the twelve benchmark applications presented in Chapter 4. I conclude the chapter with a discussion of the overarching compilation trends present across the studied benchmark applications. *Further Reading*: The unscaled and scaled ADPs presented in this chapter are the best-performing ADPs identified in Chapter 10. Chapter 7 provides an overview of how the compiler produces the unscaled and scaled ADPs from the dynamical system. Chapter 8 rigorously describes how the compiler produces unscaled ADPs and Chapter 9 rigorously describes how the compiler derives a scaled ADP from an unscaled ADP.

Chapter 7

Compilation Overview

This chapter provides a basic overview of the compiler and all of its intermediate representations. This chapter walks through an illustrative example demonstrating how the compiler maps the harmonic oscillator (Section 7.1) to the SIMPL analog device (Section 7.2). The result of this process is an analog device program (ADP) that implements the harmonic oscillator on the SIMPL hardware (Section 7.3). I choose to walk through the compilation process on a simplified hardware platform to demonstrate how each step of compilation operates more tractably.

Figure 7-1 presents a high-level overview of the compilation process. The first stage of compilation involves synthesizing an unscaled ADP which implements the input dynamical system on the target analog device (`LGraph`, Section 8). The `LGraph` compilation pass works with a specification of the dynamical system (DSS) and the analog device (ADS) and produces as output and analog device program (ADP) that implements the dynamical system on the analog device. An ADP *implements* a dynamical system if the physics of the circuit described by the ADP matches the dynamics of the target dynamical system. The circuit synthesis procedure assumes the hardware perfectly implements the input-output relations described in the ADS and is not subject to noise, quantization error, or operating- and frequency-range restrictions. The step of compilation primarily works with the input-output relations and modes described in the ADS and the differential equations described in the DSS. It produces an unscaled ADP which is later scaled by the circuit scaling procedure.

The circuit scaling procedure (`LScale`, Chapter 9) scales all the values and signals in the unscaled ADP to respect the physical constraints of the hardware. The `LScale` pass operates on the unscaled ADP produced by the `LGraph` compilation pass. It produces as output a scaled ADP which respects all the operating range and frequency constraints imposed by the analog device and minimizes the effects of noise, quantization error, and process variation on the described circuit. The `LScale` compilation pass ensures the original dynamical system simulation can be recovered from the scaled circuit by applying a compiler-derived inverting transform to the measured signals. This step of

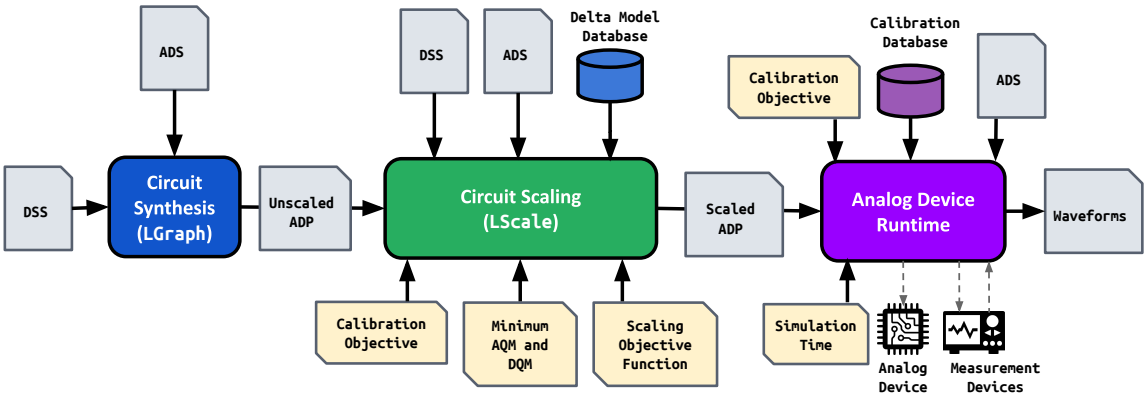


Figure 7-1: overview of compiler

compilation works the operating range, frequency, and noise annotations in the ADS and the interval annotations in the DSS. It also accepts a calibration strategy and delta model database, which together describe the process variations observed in the calibrated device on hand (Chapter 5). The scaling procedure identifies the scaling transform that minimizes the user-provided circuit metric (scaling objective function) and adheres to the user-provided minimum acceptable analog and digital quality measures (AQMmin and DQMmin). This procedure may also adjust the ADP block mode selections when necessary to produce a good scaling transform.

The scaled ADP generated by the compiler can then be executed on an analog device such as the HCDCv2 or the SIMPL analog device. The runtime accepts as input the device ADS, the scaled ADP produced by the `LScale` pass, and the execution time in simulation time units. The runtime additionally requires the end user to provide the calibration strategy that the scaling procedure used and the calibration database for the device on hand. The runtime uses these inputs to ensure the analog blocks are calibrated with the same calibration strategy used by the compiler. The runtime executes the circuit on the target analog device and returns the waveforms collected from the externally accessible pins. Note that this chapter does not explore the runtime behavior of the SIMPL analog device since it is an abstract hardware platform introduced for illustrative purposes. This chapter covers the following topics:

Simple Harmonic Oscillator: It derives the harmonic oscillator model and variable bounds from a simple mass-spring system. It introduces the harmonic oscillator mathematical model and dynamical system specification.

SIMPL Analog Device and Harmonic Oscillator ADPs: It presents the analog device specification describing the behavior for the SIMPL analog device. The SIMPL analog device is a simple reconfigurable analog computing platform loosely inspired from the HCDCv2. It then presents both the unscaled and scaled ADPs that implement the harmonic oscillator on the SIMPL analog device.

`LGraph` **Compilation Pass**: It demonstrates how the circuit synthesis procedure synthesizes an

unscaled ADP which implements the harmonic oscillator. The synthesis process first synthesizes a sub-circuit comprised of `compute` blocks which implements each differential equation in the dynamical system (Section 7.4.1). It then assembles together these sub-circuits to form a completed circuit, inserting `assembly` blocks to copy signals when necessary (Section 7.4.2). It then maps each block to a location on the SIMPL device, inserting `route` blocks to form connections when necessary (Section 7.4.3).

`LScale` Compilation Pass: It demonstrates how the circuit scaling procedure produces a scaled ADP from an unscaled ADP. The scaling process formulates the scaling problem as a constraint problem. This constraint program is then solved to produce a scaling transform. Section 7.5.1 describes how `LScale` derives a constraint problem from the scaling pass inputs.

7.1 Harmonic Oscillator

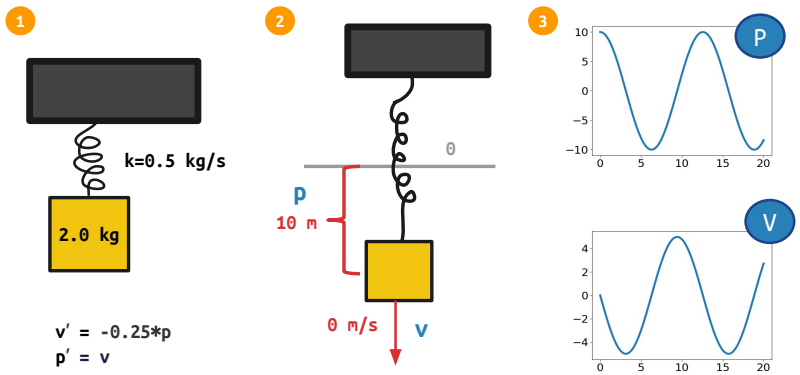


Figure 7-2: Simple harmonic oscillator

Figure 7-2 presents a spring-mass system which behaves as a harmonic oscillator. (1) This system is comprised of a 2 kg mass attached to a spring with a force constant of 0.5 kg/s. (2) I wish to model the position p and velocity v of the mass over time given an initial position of 10 m and an initial velocity of 0 m/s. (3) When released from this initial state, the the position of the mass oscillates between -10 and 10 meters and the velocity of the mass oscillates between -4 and 4 m/s. I capture the behavior of the position and velocity of the mass with the following differential equations:

$$\dot{v} = -0.25 \cdot p \qquad v(0) = 0$$

$$\dot{p} = v \qquad p(0) = 10$$

7.1.1 Harmonic Oscillator Dynamical System Specification

```
var v = integ(-0.25*p,0.0);
var p = integ(v, 10.0);
var pos = emit(p);
interval p = [-10,10];
interval v = [-5,5];
time 20;
```

Figure 7-3: Harmonic Oscillator DSS

Figure 7-3 presents the dynamical system specification (DSS) that implements the harmonic oscillator. This system models the position p and velocity v of a dampened spring oscillator over time. The `var` statements declare and define the dynamics of the p and v variables; the `interval` statements annotate each variable with the range of values it may take on. The `v` and `p` variables are annotated to be between [-5,5] m/s and [-10,10] m respectively. The `pos` variable corresponds to the observation of the position `p` over time. The oscillator executes for twenty simulation units.

The specification provides interval annotations for the position and velocity variables. These interval annotations indicate the position `p` falls between -10 and 10 m, and the velocity `v` falls between -5 and 5 m/s, respectively. These annotations can be statically derived by applying the conservation of energy principle to the system. Because this system has no dissipative forces, the potential energy and kinetic energy must always sum to the total energy of the system:

$$\texttt{Kinetic Energy} + \texttt{Potential Energy} = \texttt{Total Energy}$$

For the spring oscillator, the energy conservation equation of the system is the following:

$$\frac{1}{2}m \cdot v^2 + 1/2k \cdot p^2 = \texttt{Total Energy}$$

In its initial state, the following system has 25 Joules of energy:

$$\frac{1}{2} \cdot 2\ kg \cdot (0\ m/s)^2 + 1/2(0.5\ kg/s)(10m)^2 = 25kg \cdot m^2 s^{-2}$$

When the total energy of the system is converted entirely to kinetic energy ($\frac{1}{2}m \cdot v^2$), the velocity of the mass is either -5 or 5 m/s:

$$\frac{1}{2} \cdot 2\ kg \cdot v^2 = 25$$

When the total energy of the system is converted entirely to potential energy ($\frac{1}{2}m \cdot v^2$), the position of the mass is either -10 or 10 m.

$$\frac{1}{2} \cdot 0.5\ kg/s \cdot p^2 = 25$$

Therefore, the position and velocity of the mass fall between [-10,10] m and [-5,5] m/s, respectively. These intervals specify the *dynamic range* of the position and velocity. The dynamic range of a time-varying signal is the range of values it may assume at any point in time.

7.2 The SIMPL Analog Device

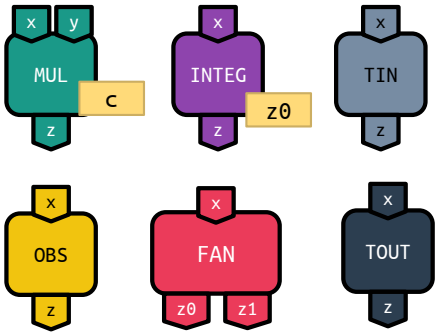


Figure 7-4: SIMPL Analog Blocks

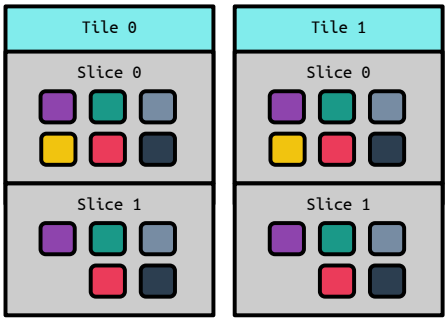


Figure 7-5: SIMPL Device Layout

This chapter targets a simplified programmable analog device inspired by the HCDCv2 analog chip. Figure 7-4 presents the six digitally programmable analog blocks available on this device. Computation blocks include integrators (`INT`), multipliers (`MUL`), and measurement blocks (`OBS`). The device also provides assembly blocks which copy analog currents (`FAN`) and routing blocks (`TIN` and `TOUT`) which direct signals inside the chip. Addition is performed by sending multiple currents to the same input port. The device has four distinct locations `(0,0)`, `(0,1)`, `(1,0)`, and `(1,1)`. The device has `TIN`,`TOUT`,`INT`, `MUL`, and `FAN` block instances at all locations. The `OBS` is only available at locations `(0,0)` and `(0,1)`. Refer to the device layout specification presented below for more information. The SIMPL analog device offers one calibration strategy `fast` which calibrates each of the blocks to minimize error. The SIMPL analog device runtime supports the elicitation of delta model parameter values from the device on hand. I next present the block specifications for each of the SIMPL blocks.

`MUL` block

The multiplier block is a compute block which accepts analog currents at input ports `x` and `y` and digital values at digital field `c`. It produces an analog current at output port `z` which implements a mode-dependent function of these inputs. The function implemented by the block depends on the block mode. I include the specification of the multiplier block below:

```
1  block mul type compute modes [(m,m,m),(x,m,m),(x,h,h),(x,h,m)] {
2     in x,y analog current; out z analog current; data c const;
3     rel z = func |(x,m,m) -> c*x |(x,h,h) -> c*x
4          |(x,h,m) -> 0.1*c*x |(m,m,m) -> 0.5*x*y
5
6     delta-par u correctable gain ideally 1.0
7     delta z = func |(x,m,m) -> u*c*x |(x,h,h) -> u*c*x
8          |(x,h,m) -> 0.1*u*c*x |(m,m,m) -> u*0.5*x*y
9
10    quantize c = linear 256; interval c = [-1,1];
11    maxfreq x = func | (*,*,h) -> 100800 | (*,*,m) -> 126000;
12    interval z = func |(*,*,m) -> [-2,2] |(*,*,h) -> [-20,20];
13    interval x = func |(*,m,*) -> [-2,2] |(*,h,*) -> [-20,20];
14    interval y = [-2,2];
15 }
```

The multiplier block works with modes (x,m,m), (x,h,h), (x,h,m), (m,m,m). The current at port `z` implements `c*x`, `c*x`, `0.1*c*x`, and `x*y` when the mode is (x,m,m), (x,h,h), (x,h,m), (m,m,m) respectively. Note that the `(x,m,m)` and `(x,h,h)` modes implement the same function but accept signals with different dynamic ranges at x and z. The multiplier block limits the the frequency of the device to 100800 when in (x,m,h) and (x,h,h) modes.

The multiplier block is subject to the effects of process variation. The delta model specification for the multiplier block has a single delta model parameter `u` which introduces a non-unity constant coefficient into the block's input-output relation. The `u` delta model parameter is correctable and implements an unexpected gain. The delta model specification implements the relation governing port `z` when `u` is one. Refer to Chapter 5 for more information on delta models, delta model parameters, and delta model specifications.

Delta Models: The delta model specification and the block instance-specific empirically

derived delta model parameter values together define the delta models for a multiplier block instance. For example, the the values for delta model parameter `u` for the multiplier blocks at locations `(0,0)` ,`(0,1)`, `(1,0)`, and `(1,1)` are presented below:

mode	MUL (0,0)	MUL (0,1)	MUL (1,0)	MUL (1,1)
(x,m,m)	0.90913	0.94875	0.9296	0.91912
(x,h,h)	0.89942	0.87330	0.85212	0.89321
(x,h,m)	0.98321	0.93110	0.94212	0.98942
(x,m,m)	1	1	1	1

The above delta model parameter values are combined with the delta model specifications to get the delta model for each mode. For example, the delta model parameter value for `u` is `0.0.98321` when the MUL `(0,0)` block instance is in `(x,h,m)` mode. According to the delta model definitions in the block specification, the delta model specification is `0.1*u*c*x` when the block is in `(x,h,m)` mode. The delta model for MUL `(0,0)` under `(x,h,m)` mode is therefore `0.1*0.98321*c*x`. This delta model is used by the compiler to more accurately target the device on hand. Refer to Section 7.5.2 for an overview of how the compiler compensates for the `u` delta model parameter.

`INTEG` block

The integrator block is a compute block which accepts an analog current at input port `x` and a digital field `z0`. It produces an output `z` which implements implements a mode-dependent function of these inputs.

```
1  block integ type compute modes [(m,m),(h,h),(h,m)] {
2     in x analog current; out z analog current; data z0 const;
3     rel z = func |(m,m) -> integ(x,2*z0) |(h,h) -> integ(x,20*z0)
4          |(h,m) -> integ(0.1*x,2*z0)
5      quantize z0 = linear 256; interval z0 = [-1,1];
6      interval z = func |(*,m) -> [-2,2] |(*,h) -> [-20,20];
7      interval x = func |(m,*) -> [-2,2] |(h,*) -> [-20,20];
8  }
```

The integrator block works with modes (m,m), (h,h), and (h,m). The analog current at port `z` implements `integ(x,2*z0)`, `integ(x, 20*z0)`, and `integ(0.1*x, 2*z0` when in

modes (m,m), (h,h), and (h,m) respectively. The integrator block behaves ideally in the presence of process variation and therefore does not define a delta model.

`OBS` block

The observation block is a compute block that accepts an analog current at input port `x` and produces a voltage at an externally acceptable output `z`.

```
1  block obs type compute modes [(dfl)] {
2     in x analog current; out z analog voltage extern;
3     rel z = emit(0.6*x);
4     interval x = [-2,2]; interval z = [-1.2,1.2]
5  }
```

The voltage at `z` implements `emit(0.6*x)`. The `obs` block has exactly one mode called `dfl`. The output port `z` is externally observable.

`FAN` block

The `fan` block is an assembly block which accepts an analog current at input port `x` and produces two copies of the current at `z0` and `z1`.

```
1  block obs type assemble modes [( m)
2  (-,-,m),(-,m),(-,m),( h),(-,-,h),(-,h),(-,h)]{
3     in x analog current; out z0,z1 analog current;
4     rel z0 = func | (*,*) -> x | (-,*,*) -> -x;
5     rel z1 = func | (*,*) -> x | (*,-,*) -> -x;
6     interval z0,z1,x = func | (*,*,m) -> -2,2 | (*,*,h) -> -20,20
7  }
```

The FAN block works with modes `(+,+,m)..(-,+,h)`. The current at `z0` is negated when in modes `(-,+,m)`,`(-,-,h)`,`(-,+,m)`, and `(-,-,h)`. The current at `z1` is negated when in modes `(+,-,m)`, `(-,-,m)`, `(+,-,h)`, and `(-,-,h)`. The analog currents at ports `x`,`z0`, and `z1` must fall within [-2,2] when in modes `(*,*,m)` and within [-20,20] when in modes `(*,*,h)`.

TIN and TOUT blocks

The tile in and tile out blocks are route blocks that send the analog current at input port x to output port z. These blocks have exactly one mode called dfl.

```
1  block tin type route modes [(dfl)]{
2     in x analog current; out z analog current;
3     rel z = x; interval z = -20,20 }
4  block tout type route modes [(dfl)]{
5     in x analog current; out z analog current;
6     rel z = x; interval z = -20,20 }
```

SIMPL Analog Device Layout Specification

```
1  device simpl {
2   freq 126000;
3   views tile, slice;
4   loc 0,1 in tile; loc 0,1 in slice;
5   blk INT,MUL,FAN,TIN,TOUT @ (*,*);
6   blk OBS @ (*,0);
7   conn INT,MUL,FAN,TIN @ (0,*) with INT,MUL,FAN,OBS,TOUT @ (0,*);
8   conn INT,MUL,FAN,TIN @ (1,*) with INT,MUL,FAN,OBS,TOUT @ (1,*);
9   conn TIN @ (0,*) with TIN @ (1,*);
10  conn TOUT @ (1,*) with TIN @ (0,*);
11 }
```

Figure 7-6: SIMPL layout specification

Figure 7-6 presents the device specification for the SIMPL analog device. The device has two views (tile and slice). There are two tiles (0 and 1) on the device, and two slices (0 and 1) per tile (line 4).

Each block location identifies the tile and slice of the block. For example, a mul block at idx(0,1) is on tile 0, slice 1 and therefore also belongs to location 0 in the tile view.

This specification attaches an integrator (INT), multiplier (MUL), copier (FAN), and two routing blocks (TIN and TOUT) to each slice (lines 5). The device has exactly two observation blocks (obs), which reside on slice 0 of tiles 0 and 1 (line 6). The SIMPL device supports

connecting all blocks to all other blocks within a tile (lines 7-8) but requires routing blocks (`TIN` and `TOUT`) to be used to connect blocks belonging to different tiles (lines 9-10). In the above specification, the baseline time constant is 126000 Hz – so that one unit of integration time corresponds to 7.93 μs (1/126000) of wall clock time. The time constant describes the integration speed of the analog device.

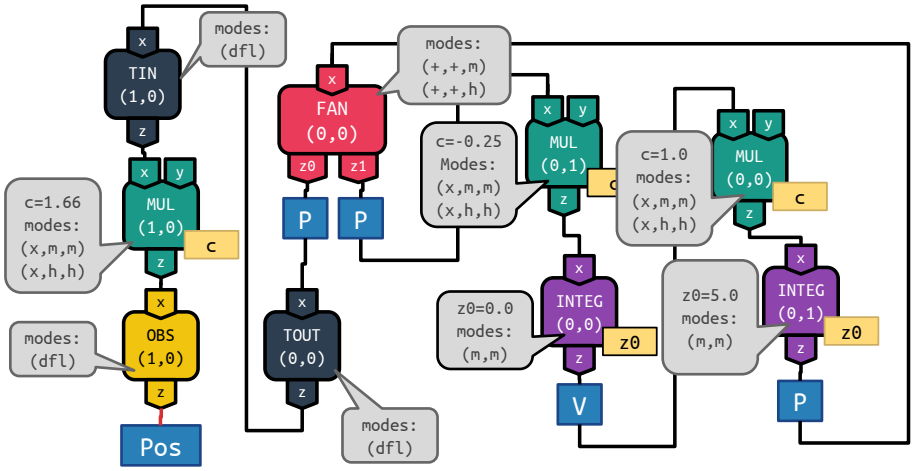


Figure 7-7: Analog circuit described by unscaled ADP

```
1  config block MUL @ (0, 1) { modes [(x,m,m),(x,h,h)]; set c at -0.25; }
2  config block INTEG @ (0, 0) { modes [(m,m)]; set z0 at 0.000; source V at z; }
3  config block MUL @ (0, 0) { modes [(x,m,m),(x,h,h)]; set c at 1.000; }
4  config block INTEG @ (0, 1) { modes [(m,m)]; set z0 at 5.000; source P at z; }
5  config block MUL @ (0, 0) { modes [(x,m,m),(x,h,h)]; set c at 1.666; }
6  config block OBS @ (1,0) { modes [(dfl)]; source Position at z }
7  config block FAN @ (0, 0) { modes [(+,+,m), (+,+,h)]; source P at x; }
8  config block TOUT @ (0, 0) { modes [(dfl)];}
9  config block TIN @ (1, 0) { modes [(dfl)];}
10 conn block INT port z loc (0, 0) with block MUL port x loc (0, 0);
11 conn block MUL port z loc (0, 0) with block INT port x loc (0, 1);
12 conn block INT port z loc (0, 1) with block FAN port x loc (0, 0);
13 conn block MUL port z loc (0, 1) with block INT port x loc (0, 0);
14 conn block FAN port z1 loc (0,0) with block MUL port x loc (0,1);
15 conn block FAN port z0 loc (0,0) with block TOUT port x loc (0,0);
16 conn block TOUT port z loc (0,0) with block TIN port x loc (1,0);
17 conn block TIN port z loc (1,0) with block MUL port x loc (1,0);
18 conn block MUL port z loc (1,0) with block OBS port x loc (1,0);
```

Figure 7-8: Unscaled ADP implementing harmonic oscillator

7.3 The Harmonic Oscillator on the SIMPL Device

I next present the unscaled and scaled ADPs that implement the harmonic oscillator on the SIMPL analog device. Refer to Chapter 6 for an overview of unscaled and scaled ADPs.

7.3.1 Unscaled ADP

Figure 7-8 presents the unscaled ADP and Figure 7-7 presents a diagram of the circuit described in the unscaled ADP. The ADP contains the following statements:

- **Block Configurations**(Line 1-9): Each block configuration instantiates all digitally settable fields present in the block (`set..` statements) and selects a a subset of modes

for the block instance (modes.. statements). the INT (0,0) and INT (0,1) are placed in (m,m) mode (Lines 2,4) and programmed so z0 equals 0.0 and 5.0 respectively. The MUL (0,0), MUL (0,1) and MUL (1,0) blocks are all placed in (x,m,m) or (x,h,h) mode and configured so the coefficient c equals 1.0, -0.25, and 1.666 respectively. The FAN (0,0) is placed in (+,+,m) or (+,+,h) mode. Note that generally speaking, each block instance in the ADP implements the same input-output relation under all the specified block modes.

Digitally Settable Connections(Lines 10-18): The ADP enables the subset of digitally programmable connections necessary to implement the circuit presented in Figure 7-7.

Dynamical System Annotations(Lines 1-9): The ADP source.. statements in Lines 1-9 indicate which analog signals implement dynamical system variables. The signal at port z of INT (0,0) implements the dynamical system variable V (line 2) and the signal at port z of INT (0,1) implements the dynamical system P (line 4). The signal at port z of the OBS (1,0) implements the observed position pos – this port is externally accessible and is observable with a measurement device.

The ADP implements the core harmonic oscillator computation with INT and MUL blocks. The ADP uses the FAN to produce multiple copies of the signal implementing the position P. This block is necessary because analog currents must be copied to be used more than once. The ADP uses the TIN and TOUT routing blocks to connect FAN (0,0) and MUL (1,0). These blocks are necessary because the MUL block is on a different tile and cannot be directly connected on the FAN block. Note that the hardware does not provide enough multipliers within a tile to implement the above configuration on a single tile.

Signal Dynamics	Dynamical System
v = integ((-0.25*p),(2*0))	v = integ(v,0.0)
p = integ((1.0*v),(2*0.50))	p = integ(-0.25*p,1.0)
pos = emit(0.60*(1.666*v))	pos = emit(p)

Figure 7-9: Unscaled dynamics of the cos benchmark

The each signal faithfully implements a dynamical system variable if the symbolic expression governing the physics of the signal matches the variable dynamics. Figure 7-9

presents a comparison of the dynamics of the ADP signals (column 1) and the dynamics of the dynamical system variables (column 2). Each signal relation contains a mixture of data field values (blue), dynamical system variables (dark blue), and block dynamics (black). Line 1 presents the dynamics of the signal implementing the velocity V at port z of INT (0,0). Line 2 presents the dynamics of the signal implementing the position P at port z of INT (0,1). Line 3 presents the signal implementing the observed position Position at port z of the OBS (1,0) block.

All of the signal relations are algebraically equivalent to their corresponding dynamical system relations. While the relations do not syntactically match up, the implement the same algebraic functions. The initial conditions for the P and V signals are scaled by 0.5 to offset the 2.0 coefficient introduced by the INT blocks. The p signal also introduces a 1.0 term which comes from MUL (0,0). In the pos signal, the unscaled ADP introduces a 1.666 term to offset the 0.60 coefficient introduced by the OBS block.

```
config block MUL @ (0, 1) {
   modes [(x,m,m)]; set c at -0.25;
   scale c = 1.6864; scale x = 0.20; scale z = 0.3200;
}
config block INTEG @ (0, 0) {
   modes [(m,m)]; set z0 at 0.000; source V at z;
   scale x = 0.3200; scale z = 0.4000; scale z0 = 0.4000;
}
config block MUL @ (0, 0) {
   modes [(x,m,m)]; set c at 1.000;
   scale c = 0.4400; scale x = 0.4000; scale z = 0.1600;
}
config block INTEG @ (0, 1) {
   modes [(m,m)]; set z0 at 5.000; source P at z;
   scale z = 0.2000; scale x = 0.1600; scale z0 = 0.2000;
}
config block MUL @ (1, 0) {
   modes [(x,m,m)]; set c at 1.666;
   scale x = 0.2000; scale z = 0.1116; scale c = 0.6002;
}
config block OBS @ (1,0) { modes [(dfl)]; source Position at z;
        scale x,z = 0.1116 }
config block FAN @ (0, 0) { modes [(+,+,m), (+,+,h)]; source P at x;
        scale x,z0,z1,z2 = 0.2000 }
config block TOUT @ (0, 0) { modes [(dfl)]; scale x,z = 0.2000; }
config block TIN @ (1, 0) { modes [(dfl)]; scale x,z = 0.2000;}
timescale 0.800;
conn block INT port z loc (0, 0) with block MUL port x loc (0, 0);
conn block MUL port z loc (0, 0) with block INT port x loc (0, 1);
conn block INT port z loc (0, 1) with block FAN port x loc (0, 0);
conn block MUL port z loc (0, 1) with block INT port x loc (0, 0);
conn block FAN port z1 loc (0,0) with block MUL port x loc (0,1);
conn block FAN port z0 loc (0,0) with block TOUT port x loc (0,0);
conn block TOUT port z loc (0,0) with block TIN port x loc (1,0);
conn block TIN port z loc (1,0) with block MUL port x loc (1,0);
conn block MUL port z loc (1,0) with block OBS port x loc (1,0);
```

Figure 7-10: Unscaled ADP implementing harmonic oscillator

7.3.2 Scaled ADP

The compiler scales the unscaled ADP to mitigate the above issues. The resulting scaled ADP respects any operating range and frequency limitations and accounts for the effects of process variation, noise, and quantization error during execution. The scaled ADP defines a scaling transform that describes how to scale all of the signals and values in the circuit. The scaling transform also defines the execution speed of the scaled computation. The scaled ADP also selects a final mode for each of the block instances.

Figure 7-10 presents the scaled ADP which implements the harmonic oscillator and Figure 7-11 presents a circuit representation of the scaled ADP. The scaled ADP specifies the scaling transform for the circuit. The scaling transform is made up of a set of magnitude scale factors `scale..` statements) for each port and data field in the circuit and a time scale factor (`timescale` statement) which specifies the execution speed of the scaled computation.

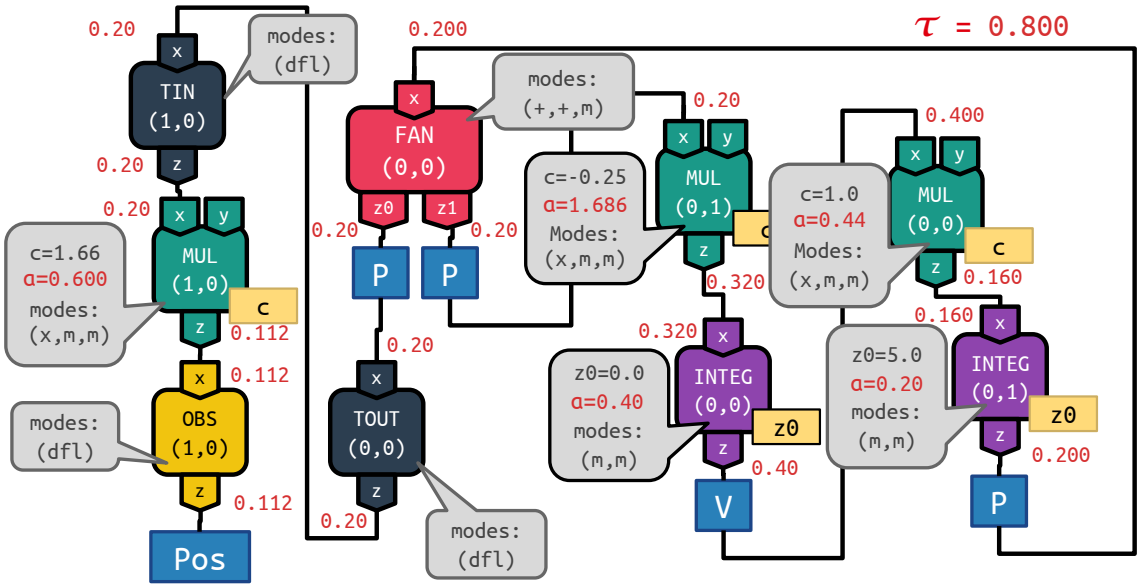


Figure 7-11: Scaled ADP implementing harmonic oscillator

Lines 3,7,11,15,21-24 define the magnitude scale factors (`scale..` statements) for each port and data field in the circuit. The scaled ADP scales the ports implementing the velocity `v`, position `p`, and observation, `pos` of the oscillator by `0.40`, `0.20`, and `0.1116` respectively. Line 25 defines a time scale factor of `0.800`. The scaled circuit slows down the implemented simulation by a factor of 0.800x the original speed. Lines 2,10, and 18 select the `(x,m,m)` mode for all three of the block instances which are in use.

Using the Scaling Transform: The scaling transform is applied before execution. The scaling transform is applied by multiplying the data fields by their respective scale factors and written to their device. For the above circuit, the `c` and `z0` data fields in the ADP are multiplied by their respective magnitude scale factors and written to the device. After execution, the end user can recover the original dynamical system trajectories from the measured by dividing the signal's amplitude by its magnitude scale factor and multiplying the time samples by the baseline integration speed and the time scale factor. For example, the original trajectory of the `pos` variable is recovered from the `pos` signal by dividing the amplitude by `0.1116` and multiplying the time samples by `0.800*126000`.

Physical Realizability: The scaled ADP presented above is physically realizable. It respects all of the operating range and frequency restrictions present in the device and attenuates away the effects of noise and quantization error:

- *Operating Range Restrictions*: The scaled ADP scales down the value at `c` from 1.66 to `0.6002*1.66=0.999` so that it falls within the operating range [-1,1]. The scaled ADP scales down the signal implementing `v` (Line 6,7) from `[-5,5]` to `0.40*[-5,5] = [-2,2]` so that it falls within the operating range [-2,2]. The ADP scales down the signal implementing `p` from [-10,10] to `0.2*[-10,10] = [-2,2]` so that it falls within the operating range [-2,2]. The ADP scales the signal implementing `pos` from `[-10,10]` to `0.1115*[-10,10]` so that it falls within the operating range `[-1.2,1.2]`

- *Frequency Restrictions*: The above circuit maps 1.25 units of hardware time to one unit of simulation time. This corresponds to a simulation speed of `0.8*126` or `100.8` kHz. Therefore, the scaled circuit's simulation speed does not exceed the maximum supported execution speed of 100.8 kHz. This frequency is the maximum frequency supported by the multipliers in the circuit.

- *Noise and Quantization Error*: The signals implementing `p` and `v` fully utilize the entire operating range of the port. The data field `z0` of `INT (0,1)` and data field `c` `MUL (1,0)` are both supplied with the maximum supported digital value `scf(0.2)*5 = scf(0.602)*1.666 = 1.0`.

  It is not usually possible to maximize every single signal and value in a given ADP. Port `z` of `MUL (0,1)` and `MUL (0,0)` both produce scaled signals with a dynamic range of `0.32*[-2.5,2.5] = 0.16*[-5,5] = [-0.8,0.8]`. These signals underutilize the available operating range of the port `[-2.0,2.0]`. The data field `c` of `MUL (0,1)` accepts the scaled value `0.44*1.0=0.44` and the data field `c` of `MUL (0,0)` accepts the scaled value `scf(1.6864)*(-0.25) = -0.4216`. Both scaled values are smaller than the maximum supported values -1 and 1.

Preservation of Original Dynamics: The scaled ADP preserves the original dynamics of the dynamical system such that the end user can recover the original dynamics from the scaled dynamics at runtime by applying an inverting transform. The scaled ADP must compensate for any empirically observed behavioral deviations in the calibrated device. The scaled ADP also compensates for any changes in the signal dynamics which arise from changing the block modes.

Signal Dynamics	Dynamical System
v_{sc} = 0.40*v	= integ(0.8^{-1}(0.94875*(1.6864*-0.25)*(0.20*p)), 2*(0.40*0.0))
p_{sc} = 0.20*p	= integ(0.8^{-1}(0.90913*(0.4400*1.0)*(0.40*v)), 2*(0.20*5.0))
pos_{sc} = 0.1116*pos	= emit(0.6*(0.9296*(0.6002*1.666)*(0.20*p))

Figure 7-12: Scaled dynamics of the harmonic oscillator

Figure 7-12 presents the dynamics of the scaled signals for the scaled ADP. The above relations contain ,agnitude and time scale factors (red) and compensation terms (grey) which capture the multiplier behavioral deviations (grey) in addition to the data fields, dynamical system variables, and the idealized block dynamics. Line 1 presents the dynamics of the signal implementing the velocity `V` at port `z` of `INT (0,0)`. Line 2 presents the dynamics of the signal implementing the position `P` at port `z` of `INT (0,1)`. Line 3 presents the signal implementing the observed position `Position` at port `z` of the `OBS (1,0)` block. Refer to Chapter 9 for more information on how the scaled signal dynamics are derived from the scaled ADP.

A scaled signal relation preserves the original dynamical system dynamics if the right hand side of the relation can be rewritten to equal the original unscaled dynamics of the signal times the signal's magnitude scale factor:

- `v` **Signal**: The magnitude scale factor 0.40 for `v` can be factored out of the derivative 0.8^{1}*0.94875*1.6864*0.20 = 0.40 and the initial value of the scaled signal implementing `v`. The expression of unfactored terms `integ(-0.25*p,2*0.0)` matches the dynamics of `v = integ(-0.25*p,0.0)`. The 0.94875 delta model parameter value is introduced into the signal dynamics when the `MUL (0,1)` block is in tx(x,m,m) mode.

- `p` **Signal**: The magnitude scale factor 0.20 for `p` can be factored out of the derivative 0.8^{1}*0.90913*0.4400*0.40 = 0.20 and the initial value of the scaled signal implementing `v`. The expression of unfactored terms `integ(1*v,2*5.0)` matches the dynamics of `p = integ(v,10.0)`. The 0.90913 delta model parameter value is introduced into the signal dynamics when the `MUL (0,0)` block is in `(x,m,m)` mode.

- `pos` **Signal**: The magnitude scale factor 0.1116 for `pos` can be factored out of the expression 0.9296*0.6600*0.20 = 0.1116 and the initial value of the scaled signal implementing `v`. The expression of unfactored terms `emit(0.6*1.666*p)` matches the

dynamics of the observed position `pos = emit(p)`. The `0.9296` delta model parameter value is introduced into the signal dynamics when the `MUL (1,0)` block is in `(x,m,m)` mode.

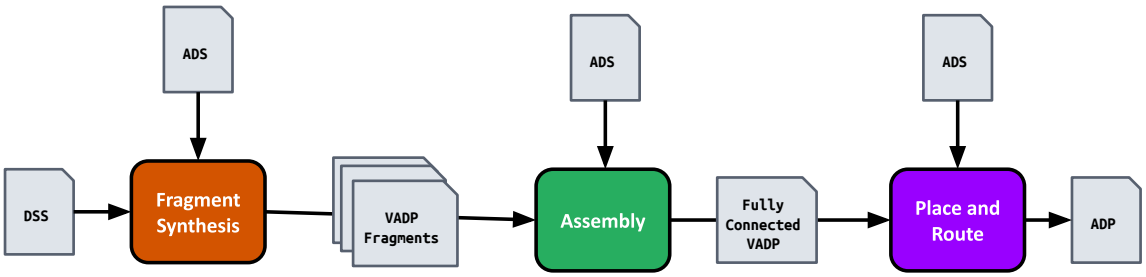


Figure 7-13: overview of circuit synthesis procedure (`LGraph`)

7.4 Circuit Synthesis (`LGraph`)

To implement a dynamical system on an analog device, the analog building blocks resident on the device must be configured and routed together so that the physics of the device (behavior of the currents/voltages over time) matches the behavior of the dynamical system. The compiler synthesizes analog device configurations that are guaranteed to be algebraically equivalent to the starting dynamical system – that is, the configuration implements a dynamical system that can be transformed into the original dynamical system by successively applying algebraic rewrite rules.

This compiler accepts as input a specification of the dynamical system to compile (DSS) and a specification of the analog device (ADS). The analog device specification describes each block's behavior and programming interface and the programmable connections available on the device. It produces, as output, an analog device program (ADP) which implements the provided dynamical system on the target analog hardware. Internally, the compiler works with an extended representation of the analog device program (vADP) that supports defining circuit fragments. This chapter covers the following:

- **Synthesis**: `LGraph` synthesizes a collection of vADP fragments made up of compute blocks that implement each relation in the dynamical system specification.
- **Assembly**: `LGraph` assembles a complete vADP that implements the dynamical system from these vADP fragments, copying and converting signals with assembly blocks when necessary. Assembly blocks perform simple operations over one input (e.g., negation) or signal conversion. This stage resolves any sinks present in the vADP fragments.
- **Routing**: `LGraph` assigns vADP block identifiers to locations on the device, insert-

ing routing blocks when necessary. The resulting ADP respects any resource and connectivity constraints present in the device.

The vADP: The vADP is the intermediate circuit representation that the `LGraph` pass uses to represent partial circuits and circuits with abstract locations. The vADP differs from the ADP in two key ways:

- **Abstract Locations**: vADP blocks are assigned abstract identifiers instead of concrete device locations. These identifiers allow the compiler to distinguish between blocks instances of the same block type. The compiler resolves these identifiers to device locations in the place-and-route stage.
- **Fragment Support**: vADPs can describe both circuit fragments and complete circuits. The vADP introduces *sink* constructs which indicate where signals implementing DSS expressions are needed in the circuit. A vADP is considered *fully connected* or *complete* if it contains no sink statements. I describe a sink as fulfilled if a signal implementing the desired dynamical system expression is provided to the sink.

Each vADP is made up of one or more vADP statements which configure and connect blocks together. These statements closely resemble the constructs provided in an analog device programming language:

- `conn(block,ident,port,block',ident',port')`: This statement connects the port `port` of the block `block` with identifier `ident` to the port `port'` of the block `block'` with identifier `ident'`.
- `config(block,ident,modes,datafield=value,..)`: This statement configures the block `block` with identifier `ident`. The block modes are set to the set of modes `modes` and the data fields are instantiated to the assigned values (`datafield=value`).
- `sink(block,ident,port,expr)`: This statement describes where an incoming signal is needed in the vADP circuit. It indicates the input port `port` of the block `block` with identifier `ident` must be provided a signal which implements the dynamical system expression `expr` to operate as expected.
- `source(block,ident,port,expr)`: This statement describes where a signal implementing an expression is being a produced. It indicates the output port `port` of the

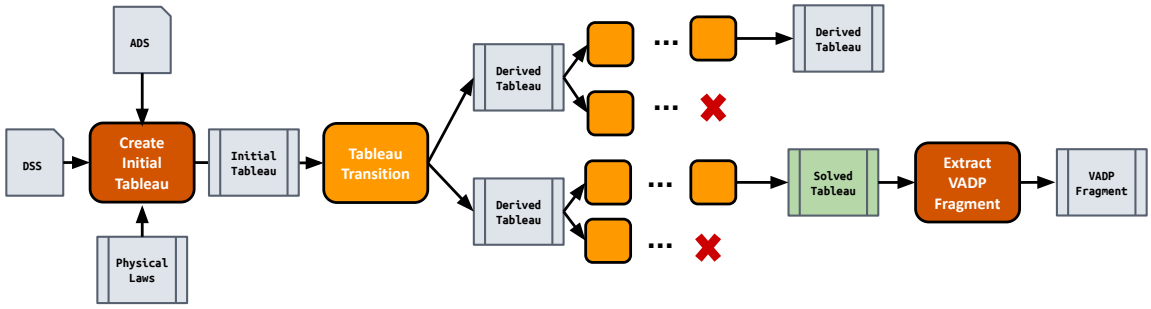


Figure 7-14: overview of tableau-based synthesis (`LGraph`)

block `block` with identifier `ident` produces a signal which implements the dynamical system expression `expr`.

7.4.1 Tableau-based vADP Fragment Synthesis

The fragment synthesis procedure accepts as input the ADS and DSS and synthesizes a vADP fragment for each variable in the DSS. The synthesis procedure invokes the fragment search algorithm on each variable-expression assignment in the DSS – this algorithm synthesizes a vADP fragment which implements the provided variable-expression assignment.

Figure 7-17 presents an overview of the fragment search algorithm. It accepts, as input, the target dynamical system relation, the available compute blocks from the ADS, and a library of physical current and voltage laws. The fragment synthesis procedure uses these laws to implement certain operations, such as addition with currents. Each physical law is a named expression describing mathematical operation implemented by the law. The expression is defined over a set of abstract law variables specific to the described physical law. Each law variable is associated with a type of signal (current, voltage) and may only work with that signal. Each physical law also comes with a simplification procedure which eliminates all occurrences of the law from a given vADP.

The fragment search algorithm performs a search over tableaus. A tableau is an algebraic representation of the state of the search. The tableau contains a set of goal dynamical system relations to implement, a set of hardware relations to use, and the vADP which has been generated so far:

- **Goals**: The tableau goals capture the parts of the circuit fragment which still need to

be synthesized. They assign dynamical system expressions to block ports (`port(block,ident,port)`) or dynamical system variables.

- **Virtual Analog Device Program**: The tableau vADP statements describe the circuit fragment which has been built up so far.

- **Hardware relations**: The tableau relations describe all of the block input-output relations the search algorithm may use to build the circuit. Each relation assigns a block port (`port(block,ident,port)`) to hardware expression. All the variables in the hardware expressions are input ports and data fields that belong to the block `block` with identifier `ident`. The tableau relations may also contain physical laws that the compiler can also use to implement computation. Each law relation assigns an invocation of a named law (`law(name,ident)`) to an expression describing the physics of the described law. All the variables in the law expression are abstract variables that are eliminated from the final vADP using the law's programmer-provided simplification procedure.

The synthesis procedure generally operates by deriving new tableaus from existing tableaus using the tableau transition relation. At a high level, the tableau translation relation selects a goal and tableau relation, finds an instantiation of the tableau relation which implements the dynamical system expression from the goal, and then extends the vADP fragment to include the configured block which implements the instantiated relation. The transition relation returns a new tableau that captures the described vADP fragment extension. Because the transition relation is non-deterministic, it can be invoked multiple times on the same tableau to produce multiple candidate search paths.

The search starts with an initial tableau with an empty vADP and a single goal – the provided dynamical system relation. The initial tableau relations contain all of the compute block input-output relations from the ADS, and the provided physical laws. It then recursively applies the tableau transition relation to the initial tableau and all subsequently derived tableaus to identify a set of solved tableaus. A tableau is considered solved when there are no goals left. The vADP fragment in the solved tableau implements the starting dynamical system relation.

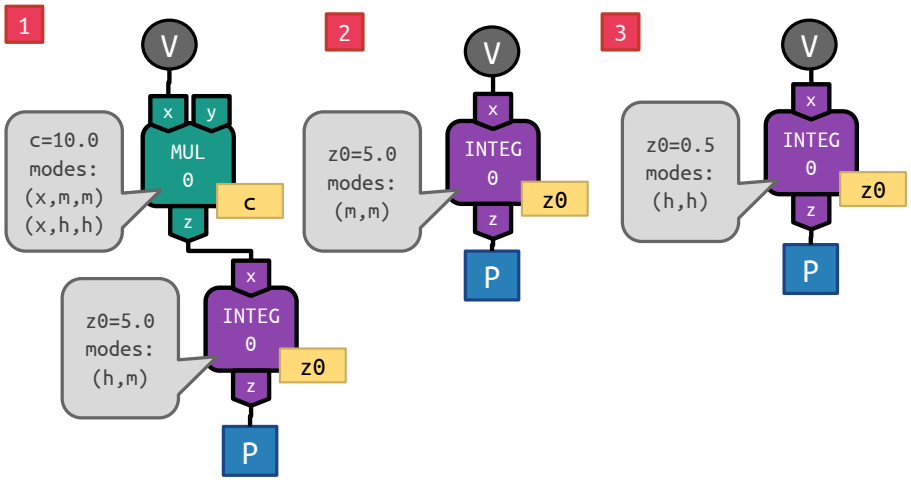


Figure 7-15: vADP fragments for harmonic oscillator position

Fragment Synthesis for Harmonic Oscillator

Figure 7-15 presents three vADP fragments which all implement the relation `p = integ(v,10)`. All of these fragments compute the position of the harmonic oscillator over time when supplied with its velocity at velocity sink `V`. I describe how each fragment implements the position of the harmonic oscillator below:

1. Fragment 1 implements the position of the harmonic oscillator with the current at output port `z` of integrator 0. It configures integrator 0 to implement `integ(0.1*x,2*5)` at port `z`, configures multiplier 0 to implement `10*x` at port `z`, and provides an analog current implementing the velocity of the harmonic oscillator to port x of multiplier 0. The current at `z` of integrator is therefore `integ(0.1*(10*v),2*5)`, which simplifies to `integ(v,10)`

2. Fragment 2 implements the position of the harmonic oscillator with the current at output port `z` of integrator zero. It configures integrator 0 to implement `integ(x,2*5)` and provides an analog current implementing the velocity of the harmonic oscillator of port x of integrator 0. The current at `z` of the integrator is therefore `integ(v,2*5)`, which simplifies to `integ(v,10)`.

3. Fragment 3 implements the position of the harmonic oscillator with the current at output port `z` of integrator zero. It configures integrator 0 to implement `integ(x,20*0.5)` and provides an analog current implementing the velocity of the harmonic oscillator of

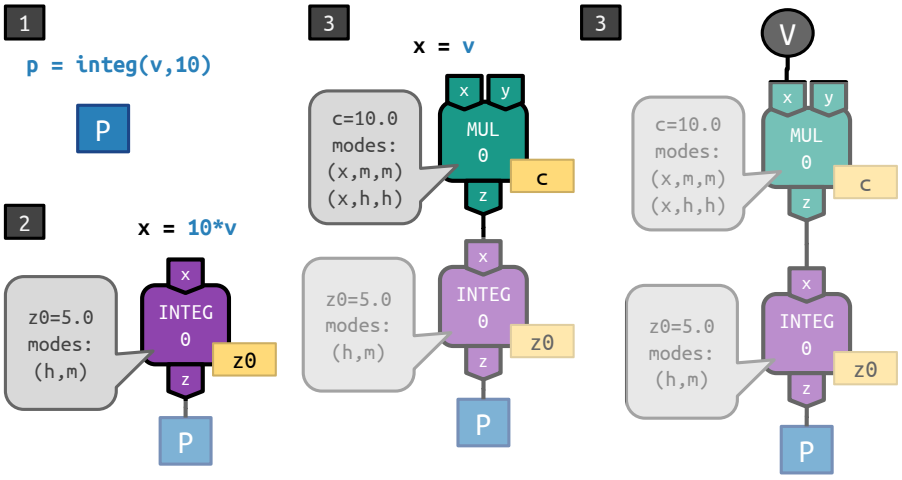


Figure 7-16: vADP synthesis procedure for position

port x of integrator 0. The current at `z` of the integrator is therefore `integ(v,20*0.5)`, which simplifies to `integ(v,10)`.

Figure 7-16 walks through the process of generating a vADP fragment (fragment 1) that implements the position of the harmonic oscillator `p=integ(v,10)`. `LGraph` first constructs an initial tableau from the dynamical system relation and the set of available compute blocks:

goals	**available relations**
`p=integ(v,10)`	`port(integ,0,z) = integ(0.1*x,2.0*z0)` if {(h,m)}
vadp	`port(integ,0,z) = integ(x,2.0*z0)` if {(m,m)}
	`port(integ,0,z) = integ(x,20.0*z0)` if {(h,h)}
	`port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}`
	`law(kirch,0) = a+b`
	...

The above tableau has a single goal that specifies the dynamics of the position of the oscillator. The initial tableau starts with an empty vADP because no fragment has been synthesized yet.

The compiler derives the available relations in the tableau from block specifications in the ADS. Each available relation specifies the dynamics implemented at a specific port of a block when the block is placed under a specific set of modes. For example, the first relation specifies the dynamics of the signal at output port `z` of the `integ` block with identifier `0`

when in `(h,m)` mode. The set of available relations also contains a single law describing Kirchhoff's law. The compiler uses Kirchhoff's law to implement addition in the produced circuits.

Kirchhoff's law states that the sum of currents at any point where wires meet is zero. The compiler may add two analog currents together in the target hardware by routing the signal from two output ports to the same input port. In this scenario, the signal flowing from the input port into the block must equal the sum of the signals coming from the output ports.

The `kirch` law describes how addition is implemented with Kirchhoff's law. In this relation, `z`, `a`, and `b` are all analog currents.The compiler uses a numerical identifier to differentiate between different invocations of Kirchhoff's law. The `kirch` law is associated with a simplification procedure (not shown) which translates a usage of Kirchoff's law to a circuit. The simplification procedure for Kirchhoff's law routes the output ports connected to law variables `a` and `b` to the input port connected to law variable `z`.

Next, `LGraph` applies the tableau transition rule to begin synthesizing the vADP fragment. The transition rule non-deterministically selects an hardware relation from the set of available hardware relations and a goal to solve. It then unifies the dynamics of the block with the target dynamical system expression to produce a configuration for the enclosing block. This invocation of the transition rule selects the output port `z` of the `integ` block with identifier `0` and mode `(h,m)`. The `integ` block implements `integ(0.1*x,2*z0)` at port `z` when in mode `(h,m)`. It rewrites the selected relation to implement the provided goal and derives the following tableau:

goals	**available relations**
`port(integ,0,x)=10*v`	`port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}`
vadp	`port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}`
`config(integ,0,{(h,m)}, z0=5)`	`port(integ,1,z) = integ(x,20.0*z0) if {(h,h)}`
`source(integ,0,z,p)`	`port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}`
	`law(kirch,0,z) = a+b`
	...

The derived tableau shown above describes the search state after the partial fragment in part (1) of the above figure is synthesized. It places the integrator with identifier `0` in

(h,m) mode and sets the digitally programmable data field to z0 (config statement). It also adds a new tableau goal which requires a signal implementing the dynamical system expression 10*v be provided to port x of this integrator to 10*v. Under this concretization, the integ block implements integ(0.1*10*v,2*5), which simplifies to integ(v,10) – the position of the harmonic oscillator. Because z0 is a digitally settable field that is instantiated when the block is programmed, it is part of the block configuration (gray speech bubble). To implement this concretization, the analog port x must be supplied with a current that models 10*v. It affixes the source label z to port z of integrator 0 – this tells the compiler that this fragment implements the position of the oscillator. This tableau removes the available hardware relation for the output port z of the block integ 0 because that output port has already been used. The tableau adds a fresh set of available hardware relations for the block integ 1 – these relations enable the compiler to use a fresh integrator later in the synthesis procedure.

LGraph next applies the tableau transition to the above derived tableau to further extend the vADP fragment to implement 10*v at input port x of integrator 0. The tableau transition function selects hardware relation describing the signal at port z of the mul block when in modes (x,m,m) or (x,h,h). This hardware relation specifies two modes because they both implement the relation c*x at output port z. LGraph determines that the current at z implements 10*v when the data field c is 10 and the input current provided to x implements the velocity v. The compiler derives the following tableau after completing the unification process:

goals	**available relations**
port(mult,0,x)=v	port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}
vadp	port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}
config(integ,0,{(h,m)}, z0=5)	port(integ,1,z) = integ(x,20.0*z0) if {(h,h)}
source(integ,0,z,p)	port(mul,1,z) = c*x if {(x,m,m),(x,h,h)}
config(mul,0,{(x,m,m),(x,h,h)},c=10)	law(kirch,0,z) = a+b
conn(mult,0,z,integ,0,x)	...

The above derived tableau describes the search state after the partial fragment in part (2) of the above figure has been synthesized. It places the multiplier with identifier 0 in (x,m,m) or (x,h,h) mode and sets data field c to 10 (config statement). It adds a single

goal requiring the harmonic oscillator velocity `v` be provided into port `x` of the multiplier. Under these conditions, the multiplier implements `10*v` at port `z` which satisfies the selected goal. The tableau also contains a vADP connection statement (`conn`) which routes the signal implementing `10*v` from the multiplier output port `z` to the input port `x` of integrator `0`.

goals	**available relations**
	`port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}`
vadp	`port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}`
`config(integ,0,{(h,m)}, z0=5)`	`port(integ,1,z)) = integ(x,20.0*z0) if {(h,h)}`
`source(integ,0,z,p)`	`port(mul,1,z) = c*x if {(x,m,m),(x,h,h)}`
`config(mul,0,{(x,m,m),(x,h,h)},c=10)`	`law(kirch,0,z) = a+b`
`conn(mult,0,z,integ,0,x)`	**...**
`sink(mult,0,v)`	

The derived tableau presented above describes the search state after the fragment in part (3) of the above figure has been synthesized. It adds a vADP sink annotation which maps the harmonic oscillator velocity (`v`) to the input port `x` of multiplier `0`.

At this point, the tableau has no goals left and is therefore considered a solved tableau. `LGraph` returns the vADP fragment in the tableau as a candidate vADP fragment that implements the position of the harmonic oscillator. The resulting fragment implements the position of the harmonic oscillator with the analog current output port `z` of integrator 0, provided an analog current carrying the velocity is supplied to input port `x`.

Example: vADP Synthesis with Physical Laws

The tableau synthesis procedure is also able to leverage physical laws to implement computation. This following example illustrates how `LGraph` uses Kirchhoff's law to implement addition. Consider the following starting tableau:

goals	**available relations**
`q=integ(2*v,10)`	`port(integ,0,z) = integ(0.1*x,2.0*z0)` if {(h,m)}
vadp	`port(integ,0,z) = integ(x,2.0*z0)` if {(m,m)}
	`port(integ,0,z) = integ(x,20.0*z0)` if {(h,h)}
	`port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}`
	`law(kirch,0) = a+b`
	...

The above tableau has a single goal (`q=integ(2*v,10)`). The tableau also provides a set of available relations implementing integration, multiplication, and addition. The addition operation is provided with by the `kirch` physical law. This physical law performs addition over two law variables (`a` and `b`). The identifier of the physical law enables `LGraph` to differentiate between multiple uses of the same law. This `kirch` relation has an identifier 0 – this indicates that the law has not been used before.

The `LGraph` synthesizer first implements the target goal using the dynamics of the integrator block in `(m,m)` mode. The resulting derived tableau describes a vADP fragment containing a single integrator block:

goals	**available relations**
`port(integ,0,x)=2*v`	`port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}`
vadp	`port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}`
`config(integ,0,{(m,m)}, z0=5)`	`port(integ,1,z) = integ(x,20.0*z0) if {(h,h)}`
`source(integ,0,z,q)`	`port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}`
	`law(kirch,0,z) = a+b`
	...

To solve the derived tableau, the synthesizer must route an analog current implementing `2*v` to the input port of the integrator. The `port(integ,0,x) = 2*v` goal encodes this requirement. The synthesizer selects the `kirch` law relation to fulfill this goal. It unifies the goal expression `2*v` with the physics of Kirchhoff's law (`a+b`) and derives a new tableau:

goals	**available relations**
`law(kirch,0,a)=v`	`port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}`
`law(kirch,0,b)=v`	
vadp	`port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}`
`config(integ,0,{(m,m)}, z0=5)`	`port(integ,1,z) = integ(x,20.0*z0) if {(h,h)}`
`source(integ,0,z,q)`	`port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}`
`conn(kirch,0,z,integ,0,x)`	`law(kirch,1,z) = a+b`
	...

The derived tableau now contains two goals which require two signals implementing `v` be mapped to the law variables `a` and `b`. The synthesizer solves these goals by affixing

vsourcestatements to the to both law variables. The resulting solved tableau contains a vADP fragment which implements q=integ(2*v,10):

goals	**available relations**
	port(integ,1,z) = integ(0.1*x,2.0*z0) if {(h,m)}
vadp	port(integ,1,z) = integ(x,2.0*z0) if {(m,m)}
config(integ,0,{(m,m)}, z0=5)	port(integ,1,z) = integ(x,20.0*z0) if {(h,h)}
source(integ,0,z,q)	port(mul,0,z) = c*x if {(x,m,m),(x,h,h)}
conn(kirch,0,z,integ,0,x)	law(kirch,1,z) = a+b
sink(kirch,0,a,v)	...
sink(kirch,0,b,v)	

This vADP fragment is incomplete in its current state because it contains a mixture of block instance ports and law variables. The synthesizer eliminates all law variables from usages of Kirchhoff's law by iteratively invoking the vADP simplification procedure which was provided with as part of the law specification. This procedure identifies a single usage of Kirchhoff's law in the vADP and translates the usage to a set of vADP connections. I present the vADP fragment with all of the law variables eliminated below:

```
source(integ,0,z,q)
config(integ,0,{(m,m)}, z0=5)
sink(integ,0,x,v)
sink(integ,0,x,v)
```

The above vADP fragment labels port x of integrator 0 with two sink statements implementing the dynamical system variable v. This fragment sends therefore two currents implementing v are sent to port x of integrator 0. Kirchhoff's law states that the sum of currents at any join point is zero. In the above circuit, the signal flowing from port x into the integrator circuitry equals the sum of the signals flowing into port x (v+v). This connection fulfills the requirement that a signal implementing 2*v be provided to the input port of integrator 0.

7.4.2 vADP Assembly

The vADP assembly step of the synthesis procedure assembles the vADP fragments computed in the fragment synthesis step to form a completed vADP. The returned completed

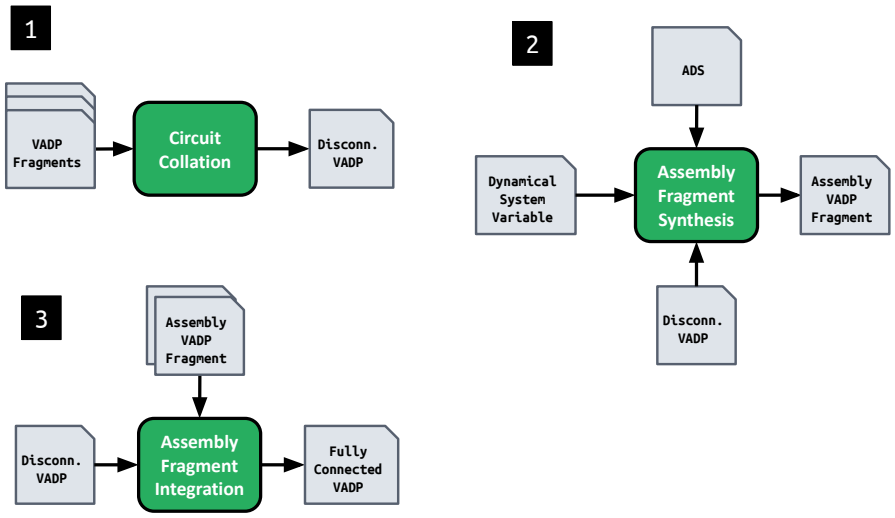


Figure 7-17: Overview of vADP assembly step.

vADP implements the dynamical system specified in the DSS. The assembly procedure inserts assembly blocks (blocks with the `assemble` type) to copy and convert signals as necessary.

The assembly step works with the collection of vADP fragments that each implement individual dynamical system relations from the DSS and the analog device specification (ADS). The analog device specification describes all the assembly blocks which may be used in the assembly process.

The assembly algorithm first combines all of the vADP fragments to form a single disconnected vADP. The vADP appears disconnected because none of the sink vADP sink statements have been fulfilled yet. The assembly algorithm first collates together the vADP fragments (circuit collation) to form the disconnected circuit. To complete the disconnected vADP, the assembly algorithm must route any generated signals (vADP `source` statements) to where they are needed (vADP `sink` statements). This is not a straightforward process because (1) some signals (analog currents) cannot directly be used more than once (2) there may be type mismatches between signal sources and sinks (3) the signal required at the sink might differ slightly from the signal provided at the source. The assembly algorithm resolves these issues by generating *assembly fragments* which link together the sources and sinks. Each assembly fragment performs all of the transformations and signal duplications necessary for a specific signal source to fulfill all related signal sinks.

The assembly algorithm next synthesizes an assembly fragment for each dynamical sys-

tem variable in the dynamical system specification. The algorithm first analyzes the disconnected circuit to determine how many copies of the target variable are needed in the completed circuit. It then produces a VADP fragment of `assembly` blocks which produces the desired set of signals.

After all of the assembly fragments have been synthesized, the assembly algorithm incorporates each generated assembly fragment into the disconnected circuit to form a completed circuit. The assembly fragment integration procedure routes the appropriate source signals to the disconnected VADP and assembly fragment sink ports. Each VADP sink is removed from the circuit once it has been fulfilled. The integration step produces a VADP with no remaining sink statements.

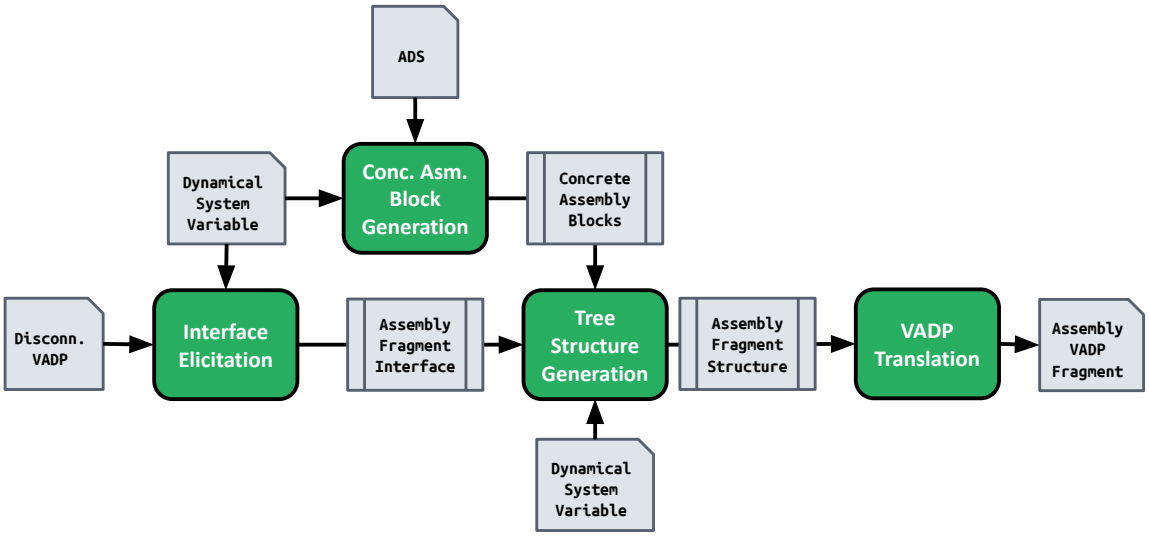


Figure 7-18: Overview of assembly fragment synthesis (`LGraph`)

Assembly Fragment Synthesis

Figure 7-18 provides an overview of how `LGraph` synthesizes an assembly fragment for a particular dynamical system variable. The assembly algorithm works with several realizations of each assembly fragment. Each of these representations works with signals are defined by the type (current, voltage, or digital) and the dynamical system expression it implements:

Assembly Fragment Interface: The assembly fragment interface specifies the input-output interface of the assembly fragment. The input interface is the collection of signals which must be provided to the fragment. The output interface is the collection of signals which are produced by the fragment. The input interface of each assembly fragment is always exactly one signal which implements a dynamical system variable.

Assembly vADP Fragment: The assembly vADP fragment is the fully realized circuit that implements the above interface. It contains exactly one vADP sink that accepts the signal at the input interface and contains one or more vADP sources, each of which implements a signal from the output interface. The circuit described by the vADP fragment is made up of configured assembly blocks arranged into a tree-like structure. These assembly blocks compute the vADP sources from the provided vADP sink.

Assembly Fragment Structure: The assembly fragment structure is a sketch of the vADP fragment that implements the fragment interface. It is an intermediate structure from which the above vADP fragment is derived. This structural representation organizes

configured assembly blocks into a roughly tree-like structure comprised of sequentially organized levels. In this structure, the level at index `i` contains all the blocks at depth `i` of the tree. Each level contains one or more configured assembly blocks and has an input and output interface. The input interface is the collection of signals required by the levels' blocks, and the output interface is the collection of signals produced by the levels' blocks. The structure has the following properties:

1. The topmost level (level 0) has an input interface that matches the assembly fragment interface.

2. Each level's output interface contains the subset of signals required at the next level's input interface. This property ensures the structure doesn't require any externally provided signals aside from the signal provided at the input interface. Signals which are used to fulfill the input interface of the next level are considered *bound*. All other signals are *free* signals which are not required to implement the structure.

3. The collection of all free signals in the structure contains the signals required at the output interface of the assembly fragment. This property ensures that the output interface is implemented by the assembly fragment.

Assembly Fragment Synthesis: The assembly fragment synthesis procedure (AFSP) accepts as input the disconnected vADP and the target dynamical system variable to target – I will call this variable the target DSS variable. First, it identifies the input-output interface of the fragment to be synthesized (interface elicitation). The assembly input interface contains one signal which implements the target DSS variable in the disconnected circuit. The assembly output interface contains a collection of signals that fulfill all the sink statements. Each of these output signals is a function of the target DSS variable provided at the input interface.

The AFSP performs a restricted form of synthesis that assumes all assembly blocks accept the target DSS variable as input. It derives a library of concrete assembly blocks from the assembly blocks in the ADS and the target DSS variable. Each concrete assembly block accepts an input signal implementing the target dynamical system variable and is assigned a fixed mode. Because the behavior of the concrete assembly blocks is deterministic, the AFSP is able to pre-compute all of the output signals produced by each block. This enables

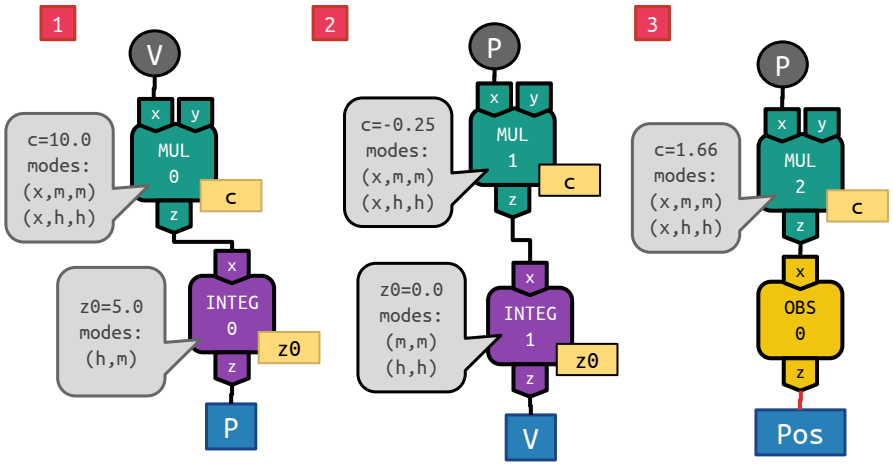


Figure 7-19: Disconnected vADP for harmonic oscillator

the AFSP to synthesize circuits without having to reason about the input-output relations implemented by the assembly blocks.

The AFSP synthesizes the fragment in two stages. It first synthesizes the assembly fragment structure and then translates the structure to a vADP assembly fragment. The fragment structure synthesis algorithm is a recursive algorithm that synthesizes the assembly fragment structure starting from the bottom-most level. At each step, the algorithm non-deterministically builds a level made up of concrete assembly blocks which implement the desired set of signals. The algorithm starts with the set of signals at the output interface and terminates when the desired set of signals is exactly the signal required at the input interface of the fragment.

Once the structure is derived, the AFSP then translates the assembly fragment structure into a vADP fragment. It converts the signal required at the input interface of the topmost level of the structure to a vADP sink. All free signals in the structure are translated to vADP sources, and all bound signals are translated to vADP connections. The blocks in the structure levels become vADP block configuration statements.

Assembly Workflow Example: Assembly for Harmonic Oscillator

The assembly stage of compilation assembles together the vADP circuit fragments, each of which implements an individual DSS variable, to form a completed circuit that implements the input DSS. The assembly procedure inserts assembly blocks to copy and convert signals as necessary.

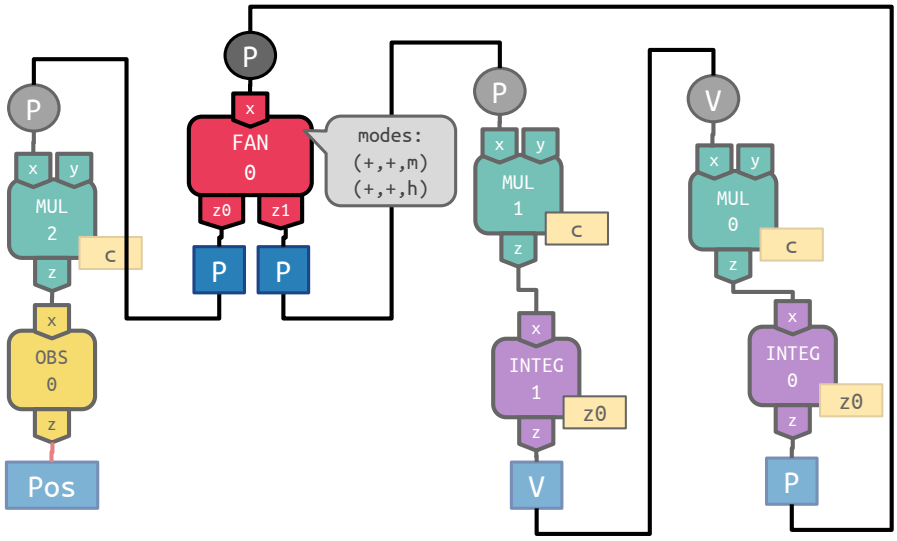


Figure 7-20: Completed vADP after assembly step.

Figure 7-19 presents the disconnected circuit. This disconnected circuit is made up of three vADP fragments, each of which implements individual relations from the DSS on the SIMPL hardware. Fragments 1 and 2 implement the position `p` and velocity `v` of the harmonic oscillator as analog currents. Fragment 3 implements the `pos` observation variable using an analog voltage. Note that the block identifiers have been reassigned so that each block is uniquely identified. To complete these fragments, an analog current carrying the position `p` must be provided to the sinks at input port `x` of multipliers 1 and 2 and an analog current carrying the velocity `v` must be provided to the sink at input port `x` of multiplier 0. The assembly procedure derives the fragment input-output interface required for each variable and synthesizes an assembly fragment if necessary:

- **Velocity (`v`)**: The velocity (`v`) variable has input interface containing one signal (`(v,current)`) and an output interface containing one signal (`(v,current)`). The assembly procedure determines the input and output interfaces match exactly and directly connect output port `z` of integrator `1` to input port `x` of multiplier `0` instead of synthesizing a fragment.

- **Position (`p`)**: The position (`p`) variable has an input interface containing one signal (`(p,current)`) and an output interface containing two signals (`(current,p)`, `(p,current)`). Because variables implemented using analog currents cannot be routed to multiple ports without compromising the integrity of the signal, analog currents

cannot be used more than once. The assembly procedure must therefore synthesize an assembly VADP fragment which duplicates the signal at the input interface. The AFSP synthesizes an assembly VADP fragment containing a fanout block with identifier `0` under modes `{(+,+,m),(+,+,h)}`. In this fragment, the input port `x` of the fanout block is a VADP sink (`sink(fan,0,x,p)`) and the output ports `z0` and `z1` of the fanout block are VADP sources (`source(fan,0,z0,p)`, and `source(fan,0,z1,p)`).

- **Observation (`pos`)**: The observed position (`pos`) has an input interface containing one signal (`(pos,voltage)`) at an empty output interface. The assembly procedure determines the observed position isn't needed anywhere does not take further action.

Figure 7-20 presents the completed, assembled VADP circuit with all of the assembly fragments integrated. In the assembled circuit, the port implementing the velocity `v` is connected to port `x` of multiplier 0 to satisfy the sink `v`. The output port implementing the position `p` is connected to the input port of the current copier 0. This current copier is part of an assembly fragment that accepts one input current implementing `p` at its input interface and produces two output currents implementing `p` at its output interface. The current copier outputs are connected to the two sinks requiring `p`. Once all the sinks in the VADP are satisfied, they are eliminated from the circuit to produce the final VADP.

Example: Synthesizing a Complex Fragment

I next describe how the AFSP synthesizes a more complex assembly fragment containing multiple levels. This fragment accepts an analog current implementing the position `p` and produces four analog currents that implement `-p`.

Figure 7-21 presents overview the concrete assembly block library derived from the ADS, the assembly fragment interface, the produced assembly fragment structure and the finalized assembly VADP fragment:

1. The concrete assembly block library contains a concrete block for each fanout block mode. Each concrete fanout block accepts a signal implementing `p` at input port `x`. The fan block produces two positive copies of `p` when in mode `(+,+,m)`, two negative copies of `p` when in mode `(-,-,m)`, and one positive and one negative copy of `p` when in mode `(+,-,m)` and `(-,+,m)`. The AFSP uses these concrete assembly blocks to synthesize the desired assembly fragment.

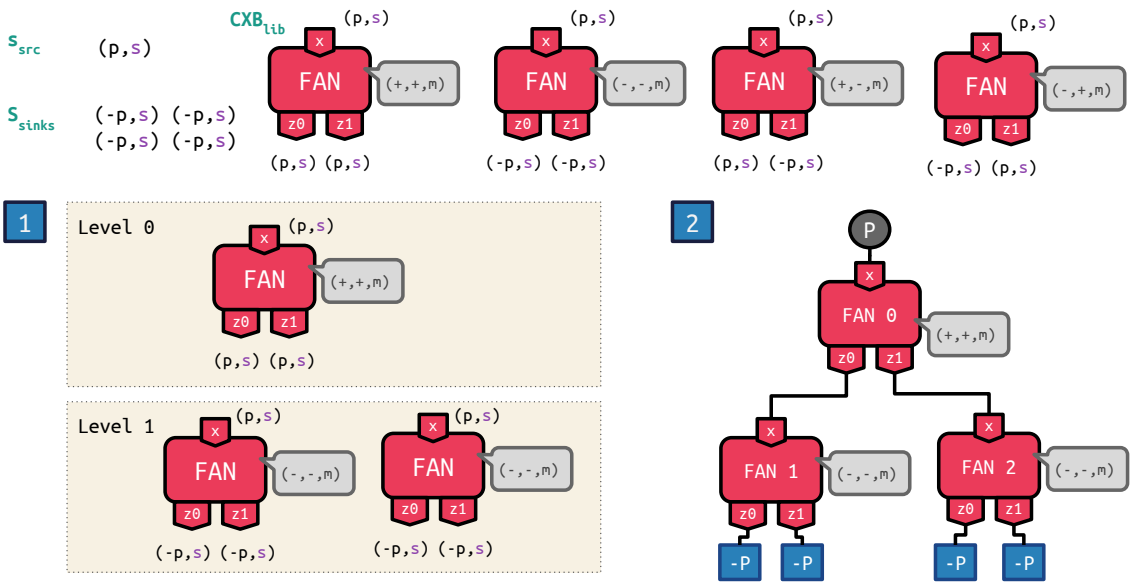


Figure 7-21: Overview of assembly fragment generation for a complex multi-level fragment. `s` is shorthand for analog currents

2. The assembly fragment interface requires one current implementing `p` at its input interface (`(p,current)`) and four currents implementing `p` at its output interface (`(-p,current)`, `(-p,current)`, `(-p,current)`, `(-p,current)`). The AFSP synthesizes a fragment which implements this interface. This interface is derived during the interface elicitation step of fragment synthesis.

3. The AFSP first derives an assembly fragment structure made up of two levels. The topmost level accepts an analog current implementing `p` and produces two analog currents implementing `p`. The bottom-most (next) level consumes the two analog currents implementing `p` from the previous level and produces four currents implementing `-p` as output.

   The assembly fragment implements the interface presented in (1) of the figure. The topmost level only requires a single signal implementing `p` input interface, and the bottom-most level produces four copies of `-p` required at the output interface. This structure also ensures that the input signals required at each level are provided by the blocks on the parent level. The concrete blocks at level 1 take two currents implementing p as input. These currents are produced as output by the concrete block in level 0.

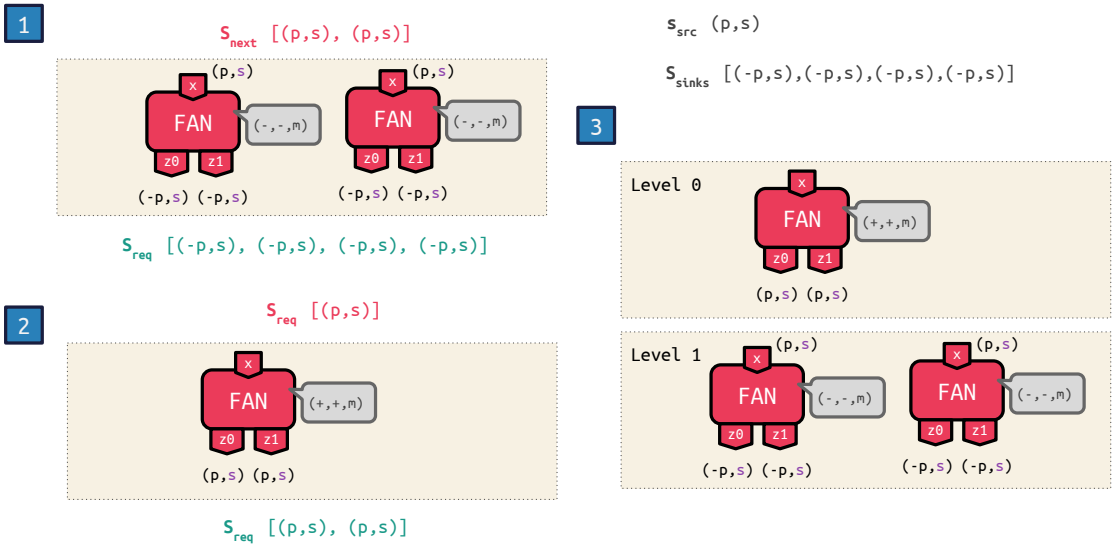


Figure 7-22: Tree structure generation. `s` is shorthand for an analog current.

4. The AFSP then translates the assembly fragment structure to a vADP fragment, inserting vADP connections, sources, and sinks when necessary. The resulting vADP fragment accepts an analog current implementing `p` at port `x` of fanout block 0 and produces four analog currents implementing `-p` at ports `z0` and `z1` of fanout blocks `1` and `2`. The signal dependences between levels become connections between block ports. The vADP sink and source statements together implement the input-output interface of the fragment.

Assembly Structure Synthesis: I next describe how the AFSP derives the assembly fragment structure from the interface presented in Figure 7-21. Each step of the structure generation procedure works with a collection of output signals (s_{req}) to generate and the source signal (s_{src}) provided to the input interface of the fragment. The structure generation procedure generates levels from the bottom-up.

It first starts by generating a level made up of concrete assembly blocks which implement the four copies of `-p` required at the output interface of the fragment. The produced level contains two fanout blocks, both of which are in `(-,-,m)` mode. This level requires two analog currents implementing `p` to produce the desired outputs.

The algorithm then recursively generates a parent level that provides two positive copies of `p`. It generates a level containing one fanout block in `(+,+,m)` mode that consumes an analog current implementing `p` and produces two analog currents implementing `p` as output.

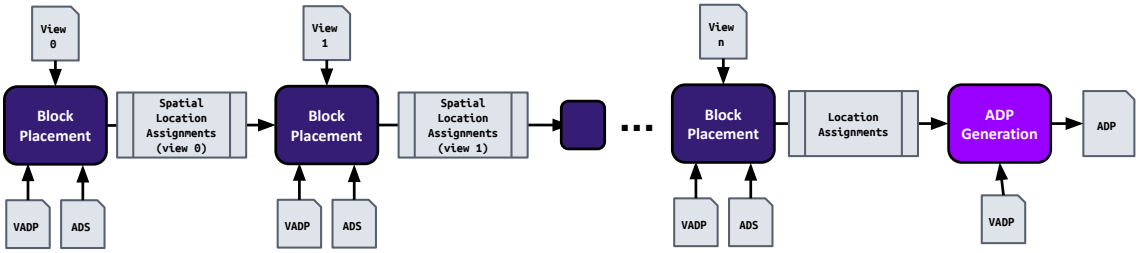


Figure 7-23: overview of vADP place and route procedure

Because this level implements the fragment's input interface (s_{src}) at its input interface, no further levels are needed. The generation procedure joins the two produced levels together and returns the combined structure.

The final generated structure is a two-level structure. The topmost level contains exactly one fanout block in `(+,+,m)` mode that accepts a signal implementing `p` and produces two signals implementing `p`. The second level consumes the two signals implementing `p` produced by the topmost level and produces four signals implementing `-p` at its output interface. The second level contains two fanout blocks in `(-,-,m)` mode.

7.4.3 vADP Place and Route

The `LGraph` place and route procedure maps vADP block instances to block locations on the target analog device and implements vADP connections with sequences of available digitally programmable connections, inserting `route` blocks as necessary. This stage of compilation inserts route blocks when a connection cannot be directly made on the analog device. This procedure works with the fully connected vADP produced in the previous compilation stage and the analog device specification. It produces as output an analog device program where all of the virtual block instances are resolved to block locations.

`LGraph` incorporates a *spatially aware* placement algorithm that places interconnected blocks close to each other in the device. It incrementally resolves the location of each block in the circuit by assigning it to spatial locations in increasingly specific views. This is necessary as the placement problem quickly becomes intractable when the locations are solved for directly. Recall that the ADS organizes device locations into sequentially organized spatial views [[1...i...n]]. Each view contains one or more spatial locations which correspond to regions where blocks are co-located on the analog device. The spatial locations on the most

specific view (n) are the unique block locations that identify block instances.

Figure 7-23 presents a high-level overview of the placement algorithm. The placement algorithm breaks down the block placement problem into several smaller, more manageable placement operations which map virtual block instances to spatial locations in the target view. Each placement operation produces a set of spatial location assignments that are guaranteed to uphold the following properties:

- **Block Availability**: If n blocks of the same type are mapped to the same spatial location, The spatial location contains at least n blocks of that type.

- **Distinct Connections**: Each connection in the vADP between two spatial locations may be mapped to a distinct path containing zero or more `route` blocks on the analog device.

The algorithm incrementally refines the spatial location of each block instance by placing it at a location in each view. The algorithm first assigns vADP block instances to spatial locations belonging to the most general (root) view. It then iteratively places the vADP block instances at spatial locations for each subsequent view, using the spatial location assignments for the parent view as restricting assignments for the placement problem. If no satisfying assignments for a view can be found, the algorithm backtracks and finds alternate assignments for a parent view. I note that this algorithm doesn't need to backtrack often, as early placement decisions assign blocks to larger structures in the device that are difficult to connect together. The produced location assignments place densely connected sets of blocks within the same spatial region of the chip. This is desirable as analog device micro-architectures typically prioritize providing programmable connections between blocks that are spatially co-located.

This algorithm continues until it has assigned the vADP block instances to spatial locations from the most specific view on the device. At this point, each spatial location corresponds to a distinct available block location on the analog device. `LGraph` then uses these spatial location assignments to generate an ADP from the input vADP (ADP generation). This step of compilation also maps vADP connections to sequences of digitally settable connections and `route` blocks on the analog device. This connection mapping procedure is guaranteed to complete successfully because the placement problem ensures

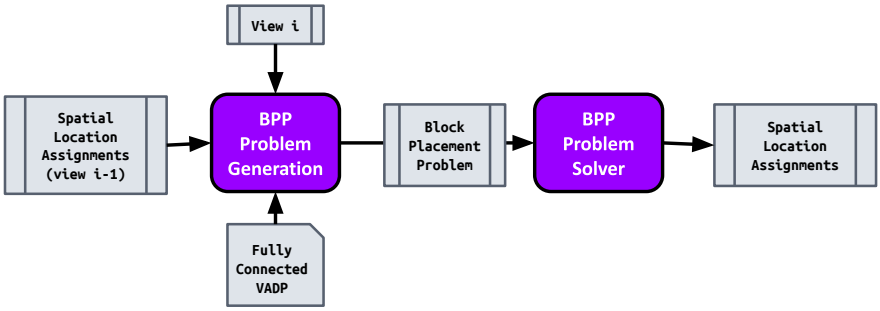


Figure 7-24: Overview of single block placement operation for spatial view i.

Block Placement

Figure 7-24 presents the workflow for a single placement operation. The placement operation accepts as input a fully connected vADP, a target spatial view, and an optional set of restricting spatial location assignments for the previous spatial view. It produces as output a set of spatial location assignments which assign the block instances from the input vADP to spatial locations from the target spatial view. The placement operation generates a BPP from the provided inputs and then solves the BPP with the BPP solver. The BPP solver uses an off-the-shelf ilp solver to find a satisfying solution and then translates the solution to a set of spatial location assignments. Note that the BPP solver is non-deterministic and can be queried multiple times to attain multiple candidate spatial location assignments.

Place and Route for Harmonic Oscillator

The routing stage of compilation maps each block identifier to a location on the analog device to produce the final ADP. `LGraph` inserts routing blocks as necessary to make all connections physically realizable. Figure 7-20 presents a completed vADP before the place and route procedure has been invoked. It has three multipliers (0,1, and 2), two integrators (0, 1), one fanout, and one observation block. Each of these block instances is uniquely identified with a numerical identifier. The place and route procedure maps each of these block instances to locations on the SIMPL analog device.

SIMPL Device Layout: Recall the SIMPL analog device specification has two spatial views: the tile view and the slice view. The tile view has two spatial locations (tx0 and `1`) and slice view has four spatial locations (`(0,0)`, `(0,1)`, `(1,0)`, and `(1,1)`). Each slice location has one of each type of block. Each tile has two incoming (`TIN`) routing blocks and

two outgoing (TOUT) routing blocks. These blocks are used to pass signals from one tile to another. A block output on tile 0 may be connected to a block input on tile 1 by routing the signal through a TOUT block on tile 0 and a TIN on tile 1. A block output on tile 1 may be connected to a block input on tile 0 by routing a signal through a TOUT on tile 1 and a TIN on tile 0. All blocks within a tile can be connected to one another without restriction.

The placement algorithm first assigns vADP block identifiers to spatial locations on the tile view. It first constructs a block placement problem (BPP) which encodes the spatial location assignment problem for the tile view. The problem is formulated to ensure any satisfying set of spatial location assignments satisfy the following criteria:

- The SIMPL analog device only provides two instances of each block type per tile. The produced assignments must therefore map at most two blocks of the same type to a single tile.
- The SIMPL analog device only provides enough routing blocks to support a maximum of four non-overlapping paths between tiles. Two of these paths move signals from tile 0 to tile 1. The other two paths move signals from tile 1 to tile 0. Therefore, the assignment problem must place blocks so that any straddling connections can be mapped to a non-overlapping path within the device.

These two constraints ensure that the computed block assignments do not exhaust the block instances and routing resources available at each tile. The produced BPP is then solved with the BPP solver to get a set of satisfying spatial location assignments. The solver computes the following assignments for the above circuit:

```
MUL 0 ↦ (0)   MUL 1 ↦ (0)   MUL 2 ↦ (1)   INT 0 ↦ (0)
INT 1 ↦ (0)   FAN 0 ↦ (0)   OBS 0 ↦ (1)
```

The above assignments map MUL 2 and OBS 0 to tile 1 and the remaining blocks to tile 0. This assignment scheme assigns at most two blocks of each type per tile and ensures only the connection between FAN 0 and MUL 2 straddles tiles. I can see this set of assignments meets the criteria listed above.

Next, LGraph maps vADP block instances to slices on the SIMPL analog device using the above tile assignments as restricting assignments. It generates a BPP for the slice view that enforce the following criteria:

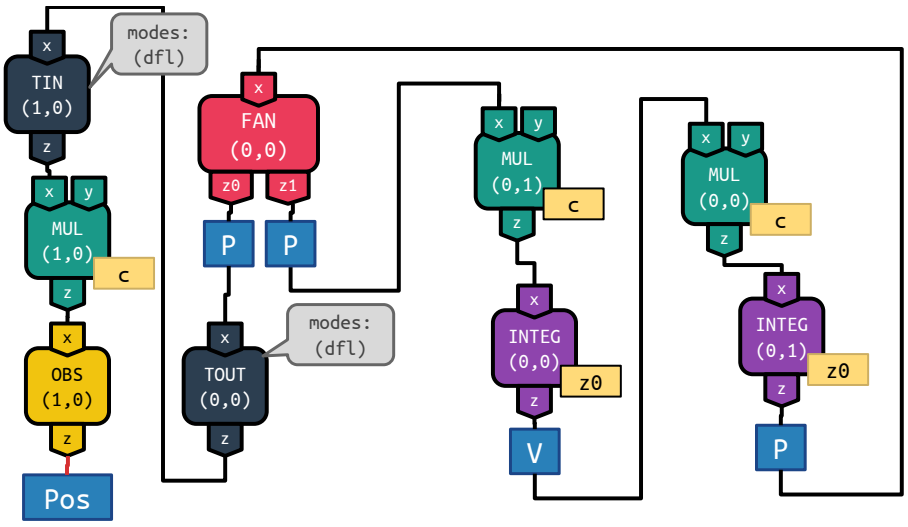


Figure 7-25: ADP of harmonic oscillator after place and route procedure.

- The produced assignments must map at most one block of each type to a single slice.
- The `MUL` 2 and `OBS` 0 blocks may only be mapped to slices (1,0) and (1,1). The remaining blocks must be mapped to slice (0,0) or (0,1). These restrictions arise because the vADP blocks have already been previously assigned to tiles. The BPP generator uses these tile assignments to restrict the space of valid spatial location assignments.

The above constraints ensure any set of produced spatial assignments does not the block instances available on each slice. The BPP also encodes connectivity constraints for the above placement problem. However, because the SIMPL analog device provides a dedicated direct connection between any pair of blocks within a tile, this does not actually constrain the mapping problem. One possible solution to the produced BPP problem is presented below:

$$\texttt{MUL}\ 0 \mapsto (0,0) \quad \texttt{MUL}\ 1 \mapsto (0,1) \quad \texttt{MUL}\ 2 \mapsto (1,0) \quad \texttt{INT}\ 0 \mapsto (0,1)$$
$$\texttt{INT}\ 1 \mapsto (0,0) \quad \texttt{FAN}\ 0 \mapsto (0,0) \quad \texttt{OBS}\ 0 \mapsto (1,0)$$

Because the slice is the most specific view in the SIMPL device, the above assignments are also valid final location assignments. `LGraph` is now free to route the connections between blocks to sequences of digitally programmable connections and routing blocks. It is able to directly map all the connections between blocks residing on the same tile to digitally settable

connections. The connection from port z of FAN (0,0) to port x of MUL 0,1 is mapped to the sequence of digitally settable connections:

FAN (0,0).z → TOUT (0,0).x → TOUT (0,0).z → TIN (1,0).x → TIN (1,0).z → MUL (1,0).x

The above connections route the current at port z of the fanout block through TOUT (0,0) and TOUT (1,0) to tile 1. The signal can then be routed to port x of MUL (1,0). Figure 7-25 presents the final unscaled ADP. In it, the vADP block identifiers have been replaced with SIMPL block locations and the relevant routing blocks (TIN and TOUT) have been introduced to make the desired connections.

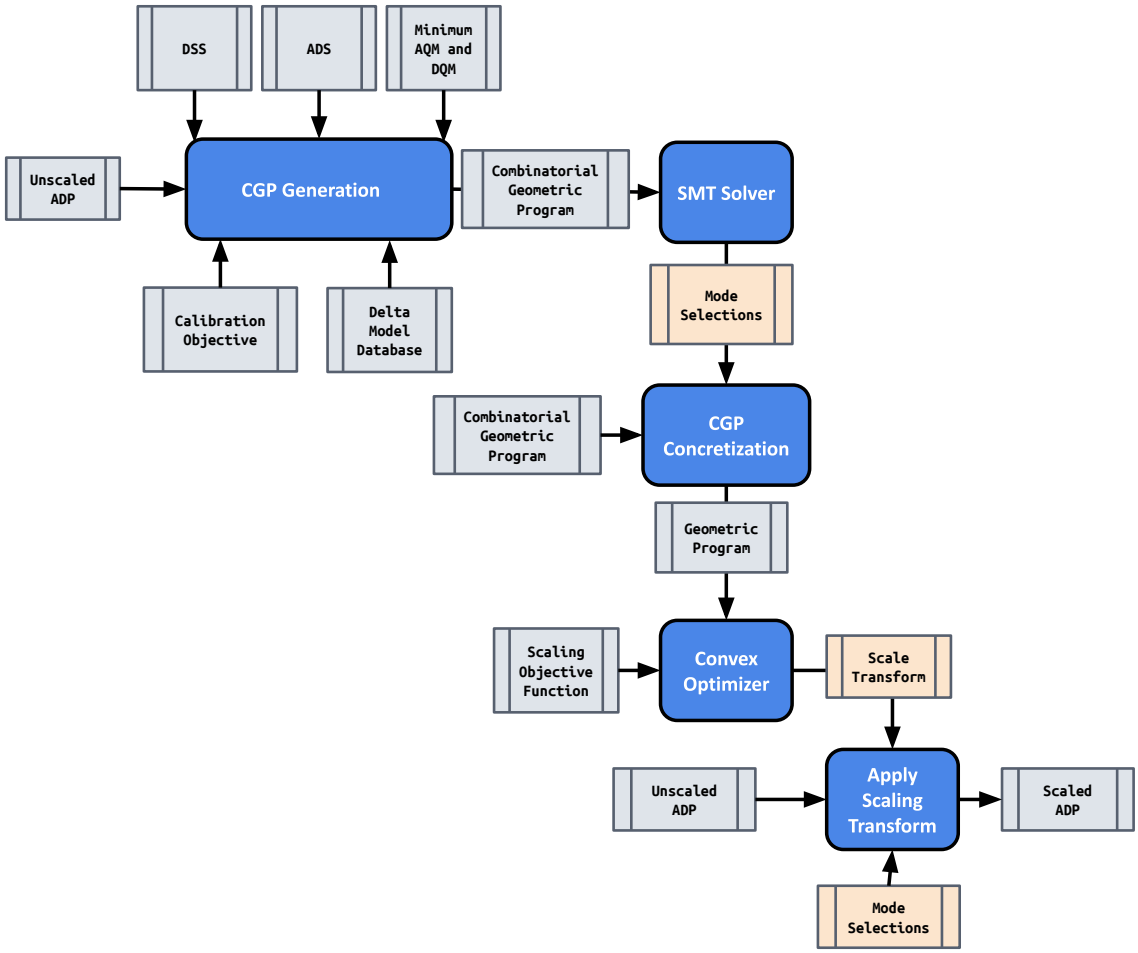


Figure 7-26: Overview of circuit scaling procedure (`LScale`)

7.5 `LScale` Compilation Pass

The `LScale` compilation scales the provided input unscaled ADP and produces a scaled ADP which is both physically realizable and recoverable. The `LScale` compilation pass formulates the problem of obtaining a physically realizable and recoverable scaling transform as a constraint satisfaction problem (specifically, a combinational geometric program). The solution to this problem delivers a set of mode selections and *scaling factors* that, when applied to the digital inputs of the analog device, produces a physically realizable, recoverable, and good quality simulation.

`LScale` Inputs

Figure 7-26 presents an overview of the `LScale` scaling procedure. `LScale` works with the following inputs:

- **Unscaled analog device program (ADP)**: The `LScale` pass accepts an unscaled ADP – this is the ADP produced by the circuit synthesis pass of compilation. The unscaled ADP implements the target dynamical system provided the hardware behaves ideally and is not subject to any low-level physical behaviors and hardware restrictions.

- **Dynamical system specification (DSS)**: The DSS specifies the target dynamical system implemented by the unscaled ADP. The DSS provides `interval` annotations that define the value ranges for the unscaled signals that implement dynamical system variables in the ADP. `LScale` uses these interval annotations to bound all of the signals in the ADP.

- **Analog device specification (ADS)**: The ADS provides a formal specification of the mode-dependent operating range and frequency restrictions and the noise and quantization error present in the hardware. The ADS also provides delta model specifications which describe which empirically derived behaviors can be compensated for through circuit scaling.

- **Calibration strategy and delta model database**: The calibration strategy, delta model database, and ADS delta model specifications together define the delta models for the device on hand. A delta model is a hardware abstraction that captures the empirically observed behavioral deviations present in a calibrated block when under a particular mode. The delta model specification defines a mathematical expression over data fields, block ports, and delta model parameters – this expression is called a parametrized delta model. The delta model parameters defined in the delta model specification capture the unexpected offsets and unexpected gains present in the hardware. Refer to Section 5.3 for information on the delta model abstraction and refer to Section 5.9 for an example of how a platform-specific runtime derives the delta model parameters for the chip on-hand.

The `LScale` pass can compensate for the subset of unexpected gains that can be tuned by scaling the block's data field values and input signals. The delta model parameters implementing these compensatable gains are annotated in the analog device specification as `correctable gain` parameters. The `LScale` pass, therefore, works with delta models that capture the correctable gains present in the device on hand.

- **Scaling objective function**: The scaling objective function specifies what circuit property the scaling procedure should minimize. The scaling objective function may, for example, maximize the execution speed of the computation or the signal-to-noise ratio of the signals in the circuit.

- **Minimum analog and digital quality measures (AQMmin and DQMmin)**: The analog and digital quality measures enforce a lower bound on the signal-to-noise and signal-to-quantization error ratios for each data field and port in the ADP. These quality measures are optional.

`LScale` Outputs

The compiler produces a scaled ADP that is optimal with respect to the user-provided scaling objective function. The `LScale` procedure derives both a scaling transform and a set of mode selections for the input ADP:

- **Mode Selections**: `LScale` selects the mode for each block instance in the ADP. `LScale` is capable of reconfiguring blocks that offer multiple equally viable modes that impose different physical limitations.

- **Scaling Transform**: The scaling transform describes how all the values in the ADP are scaled. It contains magnitude scaling factors that scale the data fields and signals in the circuit and a time scaling factor that scales the simulation speed of the circuit.

  The scaling transform identifies the recovery transform for the scaled circuit. The recovery transform recovers the original dynamical system dynamics from any measured signal in the scaled circuit. This transform recovers the magnitude of the signal by dividing the amplitude of the signal by the magnitude scale factor of the associated port. It recovers the time of the signal by multiplying the time samples, measured in

wall-clock seconds, by the hardware time constant from the ADS (measured in Hz) and the time scale factor of the scaled ADP.

The compiler incorporates the computed scaling transform and the mode selections into the input unscaled ADP to produce the scaled ADP. The scaled ADP is the final output of the `LScale` pass.

The scaled circuit contains all the information necessary to apply the scaling transform to the circuit and recover the original dynamical system trajectories from the scaled signals. At execution time, the runtime system applies the scaling transform to the circuit by multiplying each constant data field by its respective magnitude scale factor. After execution, the runtime uses the recovery transform to recover the original dynamical system trajectories from the measured signals. Refer to Chapter 6 for a more detailed discussion of scaled and unscaled circuits.

General Operation

The `LScale` pass formulates the problem of identifying a scaling transform and set of mode selections as a CGP. The compiler derives a combinatorial geometric program (CGP) from the ADP, ADS, DSS, the optional AQMmin and DQMmin, the delta model database, and the calibration strategy.

The `LScale` pass solves the CGP to obtain a set of mode selections and a scaling transform that optimally scales the circuit. The `LScale` pass identifies the scaling transform that minimizes the user-provided scaling objective function under the chosen set of mode selections.

A CGP is a type of constraint problem made up of both integer-valued variables and positive, constant, real-valued variables. The CGP contains both combinatorial constraints over integer and real variables and geometric programming constraints over real variables. The compiler directly solves the CGP with an SMT solver to produce a set of integer-valued and real-valued variable assignments.

The `LScale` pass captures the selected mode for each block instance with integer-valued variables and the scaling transform and mode-dependent hardware properties with real-valued variables. `LScale` encodes all of the constraints that ensure the scaling transform is physically realizable, of good quality, and recoverable as geometric programming constraints.

The SMT constraints encode the effect of different block mode selections on the scaling problem.

Resolving the integer-valued variables to values eliminates all of the combinatorial constraints from the CGP and reduces the CGP to a geometric programming problem (GP). The remaining geometric programming constraints and real-valued variables form a geometric programming problem (GP). A geometric programming problem (GP) is a type of convex optimization problem that can be optimally and efficiently solved with a numerical solver. Refer to Chapter 9 for a rigorous description of the CGP and GP.

`LScale` first solves the CGP with an SMT solver to obtain a set of mode selections – the CGP represents these mode selections as integer-valued variable assignments. The compiler queries the SMT solver repeatedly to get multiple viable mode selections for the provided ADP. The compiler then applies these mode selections to the CGP to produce a geometric programming problem (GP). `LScale` then solves the GP with a convex solver using the user-provided scaling objective function as the minimization criteria. The resulting solution is optimal with respect to the provided scaling objective function. The compiler then incorporates the derived scaling transform and the computed mode selections into the input unscaled ADP to produce the scaled ADP.

7.5.1 CGP Generation Procedure

In this section, I cover how `LScale` generates the combinational geometric programming problem (CGP) from the compilation pass inputs. The CGP generation procedure produces a CGP for the input ADP which can be solved to obtain a scaling transform and set of mode selections. The CGP generation procedure produces many kinds of constraints:

Operating Range Constraints: The ports and data fields in the analog hardware impose operating range restrictions on the supplied signals and values. The operating range constraints ensure that each scaled signal and value falls within the operating range of the data field or port. `LScale` derives these constraints by inspecting the operation range limitations specified by the ADS. It derives the interval of the unscaled signal by analyzing the ADP with the interval annotations provided in the DSS.

Frequency Constraints: Some analog blocks impose frequency limitations on the mapped computation. The frequency constraints limit the simulation speed of the scaled circuit.

`LScale` derives these constraints from the frequency limitations specified in the ADS.

Analog and Digital Quality Restrictions: In the hardware, the fidelity of the analog signals is affected by analog noise. The fidelity of the digital signals is affected by quantization error. The quantization error and noise constraints ensure that `LScale` does not scale down any single value or signal to the point where noise and quantization error compromises signal fidelity.

These constraints work with a proxy for signal or value quality called a quality measure. The quality measure of a scaled signal is the ratio of the magnitude of the scaled signal to the noise or quantization error associated with the analog port or digital data field. The noise constraints ensure each analog signal has a quality measure that exceeds the user-provided AQM. The quantization constraints ensure each digital signal and value has a quality measure that exceeds the user-provided DQM.

Connection Constraints: The connection constraints ensure the magnitude scale variables assigned to two connected ports are equal. These constraints ensure `LScale` scales each pair of connected ports by the same amount.

Factor Constraints: The factor constraints ensure that a positive, constant, scale factor may be factored out of the idealized dynamics of each output port in the ADP. These constraints ensure the scaling transform can be propagated through the dynamics of each block and separated from the original idealized dynamics at each output port. The computed scale factors must also compensate for any unexpected gains in the block instance. An unexpected gain is a type of empirically observed behavioral deviation which can be compensated for by scaling the circuit. The `LScale` pass produces the factor constraints for a particular block instance with a specialized analysis called the factor constraint generation procedure. Refer to Section 7.5.2 for more information on the factor constraint generation procedure.

Illustrative Example: Harmonic Oscillator

In the following section, I walk through how `LScale` generates each of the above constraints for the above unscaled ADP implementing the harmonic oscillator.

Operating Range Constraint: `LScale` generates an operating range constraint for each port and data field in the above ADP. Each operating range constraint ensures the scaled analog signal or value falls within the operating range of the port or data field:

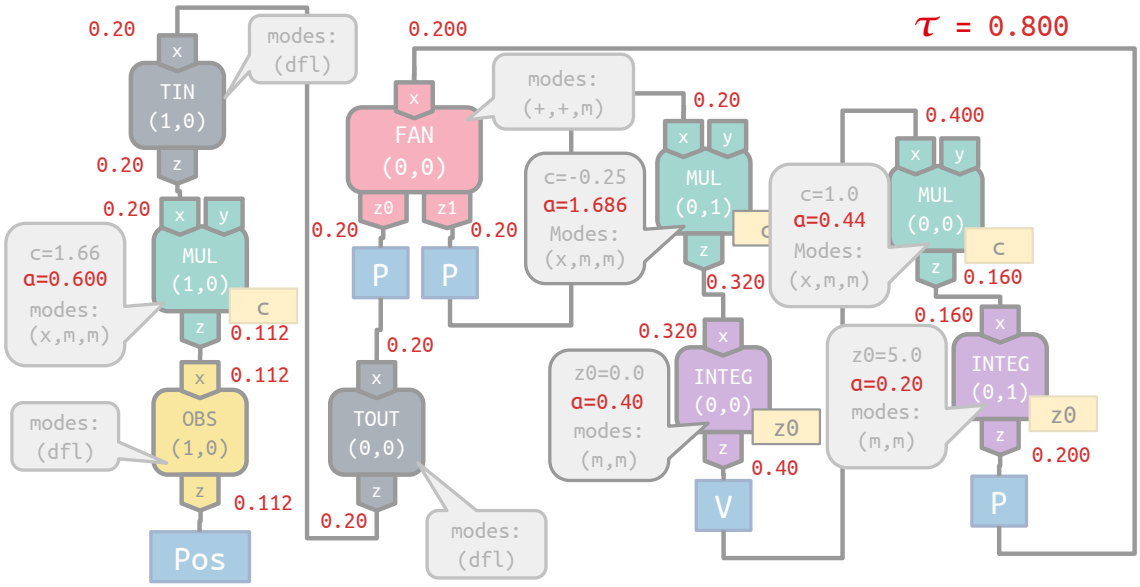


Figure 7-27: Scaled ADP implementing harmonic oscillator.

$$\texttt{mag}(\texttt{INT(0,0)},\texttt{x}) \cdot [-2.5, 2.5] \subseteq \texttt{op-range-prop}(\texttt{INT(0,0)},\texttt{x}) \cdot [-2, 2]$$

The above constraint ensures the dynamic range of the unscaled signal at port `x` ([-2.5,2.5]) of integrator `INT (0,0)` falls within the operating range of port `x` when scaled by the port's magnitude scale factor $u(\texttt{INT(0,0)}, x)$. The `op-range-prop(INT(0,0),)` property is a pair of mode-dependent property variables which captures the variations in operating ranges across block modes:

$$\begin{aligned}
\texttt{mode}(\texttt{INT(0,0)}) = \texttt{(m,m)} &\implies \texttt{op-range-prop}(\texttt{INT(0,0)},\texttt{x}) = [1, 1] \\
\texttt{mode}(\texttt{INT(0,0)}) = \texttt{(m,h)} &\implies \texttt{op-range-prop}(\texttt{INT(0,0)},\texttt{x}) = [1, 1] \\
\texttt{mode}(\texttt{INT(0,0)}) = \texttt{(h,h)} &\implies \texttt{op-range-prop}(\texttt{INT(0,0)},\texttt{x}) = [10, 10] \\
\texttt{mode}(\texttt{INT(0,0)}) = \texttt{(h,m)} &\implies \texttt{op-range-prop}(\texttt{INT(0,0)},\texttt{x}) = [10, 10]
\end{aligned}$$

The above integer implication constraints capture the relationship between the integrator mode and the operating range of port `x`. The `mode(INT(0,0))` integer variable encodes the mode selected for `mode(INT(0,0))`. When the mode is `(m,m)` or `(m,h)`, the operating range of port `x` is $[1, 1] \cdot [-2, 2]$. When the mode is `h,h` or `(h,m)` the operating range of port `x` is $[10, 10] \cdot [-2, 2] = [-20, 20,]$.

`LGraph` also generates operating range constraints for constant signals and data fields:

$$\texttt{mag}(\texttt{INT(0,1)}, z0) \cdot [5,5] \subseteq \texttt{op-range-prop}(\texttt{INT(0,1)}, \texttt{z0}) \cdot [-1,1]$$

The above constraint ensures data field providing the the scaled initial condition for integrator `INT(0,1)` falls within $[-1,1]$. The operating range property `op-range-prop(INT(0,1),z0)` always resolves to [1,1].

Analog Quality Restrictions: `LScale` produces analog quality constraints for each analog signal in the ADP. These constraints ensure that all analog signals are not scaled down to the point where they are overtaken by noise. `LScale` ensures the best-case signal to noise ratio for each scaled signal exceeds the analog quality measure (AQM) for the scaling problem:

$$\frac{\texttt{mag}(\texttt{INT(0,0)}, \texttt{x}) \cdot 2.5}{\texttt{noise-prop}(\texttt{INT(0,0)}, x)} \geq \text{AQM}$$

The above constraint ensures the ratio of the the magnitude of the scaled signal at port `x` of integrator `INT(0,0)` to the noise at that port `noise-prop(INT(0,0),x)` exceeds the real-valued analog quality measure variable (AQM). The noise at port `x` of the integrator block is dependent on the selected mode. It resolves to `0.001` when in `(m,m)` or `(m,h)` mode and resolves to `0.01` when in `(h,m)` and `(h,h)` mode. `LScale` adds a second constraint ensuring that the AQM of the scaled circuit exceeds the user-provided minimum analog quality measure:

$$\text{AQM} \geq \text{AQM}_{\text{MIN}}$$

The objective scaling function employed by the compiler often incorporates the quality measure variables into the objective scaling function to maximize the signal quality. Maximizing the AQM, for example, maximizes the quality measure of the worst-quality analog signal in the ADP.

Digital Quality Restrictions: `LScale` produces digital quality constraints which ensure each digital signal and data field value in the ADP are not scaled down to the point where they are overtaken by quantization error:

$$\frac{\texttt{mag}(\texttt{INT(0,1)}, \texttt{z0}) \cdot 5.0}{\texttt{quant-prop}(\texttt{INT(0,1)}, z0)} \geq \text{DQM}$$

The above constraint ensures the ratio of the scaled value written to data field `z0` of block `INT(0,1)` to the quantization error `quant-prop(INT(0,1)`$z0$,) clears the digital quality measure for the circuit. The quantization error is derived from the `quantize statements` in the ADS block specification. The quantization error for the above data field is always `1/128 = 0.0078125`. `LScale` adds a second constraint ensuring the DQM of the scaled circuit exceeds the user-provided digital quality measure:

$$\text{DQM} \geq \text{DQM}_{\text{MIN}}$$

Like the AQM variable, the DQM variable may also appear in the scaling objective function. Maximizing the DQM maximizes the quality measure of the worst-quality digital signal or value in the ADP.

Frequency Constraints: `LGraph` imposes frequency constraints that limit the speed of the scaled simulation. These constraints are sometimes necessary to use blocks that deviate from their specification when supplied signals with high-frequency components. The multiplier block `MUL(0,0)` requires the simulation be run at a maximum speed of 100.8 Khz when in `(h,h)` or `(m,h)` mode:

$$\texttt{time-var} \leq \texttt{max-freq-prop}(\texttt{MUL(0,0)}, z)$$

The above constraint ensure the time scaling factor `time-var` doesn't exceed the maximum supported speed of the multiplier block. The `max-freq-prop`$(\texttt{MUL(0,0)}, z)$ property resolves to $\frac{100.8kHz}{126khz} = 0.8$ when the mode selection variable `mode`$(\texttt{MUL(0,0)})$ is set to `(x,h,h)` or `(x,m,h)`. It resolves to `1.0` otherwise.

Connectivity Constraints: `LScale` produces connectivity constraints that ensure the signals at two connected ports are scaled by the same amount. This constraint is necessary for ensuring the original simulation is recoverable:

$$\texttt{mag}(\texttt{MUL(0,1)}, \texttt{z}) = \texttt{mag}(\texttt{INT(0,0)}, \texttt{x})$$

The above connectivity constraint ensures the signal at port `z` of block `MUL(0,1)` is scaled by the same amount as the signal at port `z` of block `INT(0,0)`.

Factor Constraints: `LScale` produces factor constraints that ensure the scaling transform can be eliminated from each signal by applying the compiler-derived recovery transform. The factor constraints also ensure that the scaling transform compensates for any unexpected gains present in the device on hand.

The factor constraints ensure that the scaled dynamics at each output port equals the original dynamics of the signal times a constant scaling factor. This constant scaling factor is the magnitude scale factor of the port. For example, the following expression describes the scaled dynamics of the signal at output port `z` of multiplier `MUL (0,1)`:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot (\texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{c}) \cdot (\texttt{mag}(\texttt{MUL(0,1)}, \texttt{x}) \cdot \texttt{x})$$

The above scaled expression contains the port and data field variables `c` and `x`, the magnitude scale factors for ports `c` and `x` and the process variation variable property which captures the

empirically observed variations in behavior. The above process variation variable resolves to 0.95, 0.87, 0.093, and 8.9 when the multiplier block is in `(x,m,m)`, `(x,h,h)`, `(x,h,m)`, and `(x,m,h)` respectively. I wish to separate out the original, idealized dynamics of the output port `z` (`c*x`) from the above expression. This can be accomplished by reshuffling the magnitude scale variables and process variation variable properties in the above expression:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x}) \cdot (\texttt{c} \cdot \texttt{x})$$

The magnitude scale factor at port `z` of multiplier `MUL(0,1)` must therefore equal the above expression of scale factors:

$$\texttt{mag}(\texttt{MUL(0,1)}, \texttt{z}) = \textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x})$$

Factor Constraints and Integration: In some cases, `LScale` may produce additional factor constraints to ensure an expression of magnitude and time scale factors and properties can be factored out of the scaled dynamics of a block. Take for, for example, the integrator block `INT(0,0)` which implements the following scaled dynamics:

$$\begin{aligned} z \quad &= \int \textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 0) \cdot (\texttt{mag}(\texttt{INT(0,0)}, \texttt{x}) \cdot \texttt{x}) \cdot \texttt{time-var}\ dt \\ z(0) \quad &= \textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 1) \cdot \texttt{mag}(\texttt{INT(0,0)}, \texttt{z0}) \cdot \texttt{z0} \end{aligned}$$

The above integration operation scales hardware integration time by the time scale factor `time-var` and scales the input signal `x` and initial condition by their respective magnitude scale variables. Both the derivative signal and initial condition of the integrator are scaled by empirically observed mode-dependent constant coefficients $\textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 0)$ and $\textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 1)$. For the above relation to faithfully integrate the desired signal, the magnitude scale factor for the initial value of the signal at port `z` ($z(0)$) must match the magnitude scale factor at signal `z`. `LScale` therefore produces the following intermediate constraint which enforces this requirement:

$$\begin{aligned} \texttt{mag}(\texttt{INT(0,0)}, \texttt{z}) \quad &= \textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{INT(0,0)}, \texttt{x}) \cdot \texttt{time-var} \\ &= \textit{proc-var}(\texttt{INT(0,0)}, \texttt{z}, 1) \cdot \texttt{mag}(\texttt{INT(0,0)}, \texttt{z0}) \end{aligned}$$

The above constraint ensures the factored scale expression for both the initial condition $z(0)$ and the variable z match. This factored scale expression must equal the magnitude scale factor assigned to port `z`.

Factor Constraints and Expression Data Fields: The CGP generation procedure modifies the expressions mapped to expression data fields to more flexibly scale the circuit. This capability of the `LScale` procedure does not naturally appear in the harmonic oscillator example presented in

this chapter because the harmonic oscillator dynamical system does not define any uninterpreted functions, and the SIMPL analog device does not offer any analog blocks that contain expression data fields. For completeness, I will briefly illustrate this concept using the programmable user-defined function block `lut` introduced in Section 5.7.

The `lut` block contains a digital output port `z`, a digital input port `x`, and a programmable expression data field `f` that accepts one input. The output port `z` of the `lut` block computes `f(x)`. For ADPs containing expression data fields, the `LScale` pass introduces real-valued injection variables that introduce coefficients into the expressions mapped to expression data fields. I summarize the original, idealized dynamics and the scaled dynamics with the injection variables incorporated for the `lut` at `(0,0)` below:

$$z \quad = \texttt{inj}(\texttt{lut(0,0)},\texttt{f},1)\cdot \texttt{f(inj}(\texttt{lut(0,0)},\texttt{f},0)\cdot\texttt{mag}(\texttt{lut(0,0)},\texttt{x})\cdot\texttt{x)}$$

The `LScale` pass introduces the `inj`(`lut (0,0)`,`f`,0) injection variable to scale the input argument to the expression data field and the `inj`(`lut (0,0)`,`f`,1) injection variable to scale the result of the expression data field. These injection variables are resolved to values during optimization and incorporated into the expression mapped to the expression data field. The `LScale` pass introduces the following constraints into the CGP:

$$\begin{aligned}\texttt{mag}(\texttt{lut(0,0)},\texttt{z}) \quad &= \texttt{inj}(\texttt{lut(0,0)},\texttt{g},1)\\ 1 \quad &= \texttt{inj}(\texttt{lut(0,0)},\texttt{g},0)\cdot\texttt{mag}(\texttt{lut(0,0)},\texttt{x})\end{aligned}$$

The above constraints eliminate the scaling transform from the data field inputs and scale the data field result. The first constraint scales the data field result, and the second constraint eliminates the scaling factor from the data field input `x`. Because expression data fields are often mapped to highly nonlinear expressions, it is often desirable to eliminate the scaling transform from the data field inputs as it removes the need to propagate the scaling transform through the mapped expression. Refer to Chapter 9 for a detailed discussion on injection variables and expression data field constraints.

7.5.2 Factor Constraint Generation

The factor constraint generation procedure automatically produces the factor constraints described in the previous section. It accepts as input the block instance, block configuration (from the ADP), and output port to analyze, the delta model database and calibration strategy to target, and the device ADS. The factor constraint generation procedure internally works with a symbolic expression, called a master expression, which models all unexpected gains which can be corrected for through

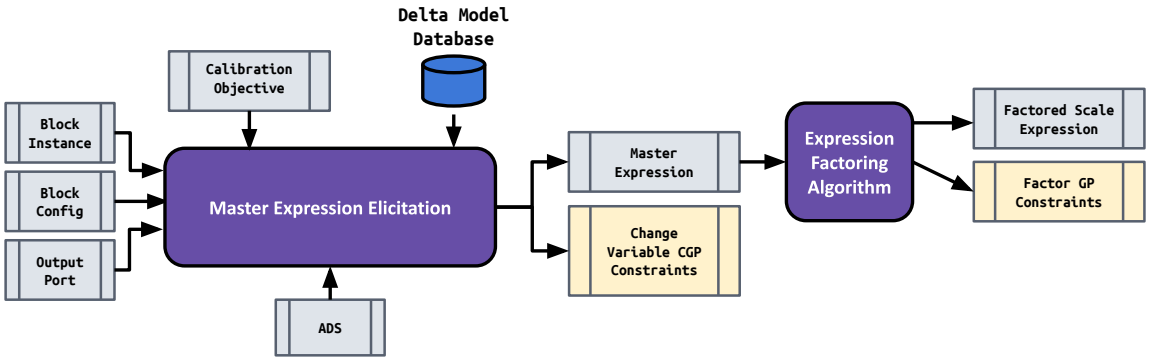


Figure 7-28: Overview of the factor constraint generation procedure.

circuit scaling and captures the effect of changing the block mode on the computation. The factor constraint generation procedure first elicits a master expression from the provided inputs and then generates the factor constraints from the derived master expression. The factor constraint generation routine produces, as output, a set of CGP and GP constraints and an expression of scale transform variables and process variation property variables which has been factored out of the scaled master expression.

In this section, I describe the operation of the factor constraint generation procedure for the for output port `z` of multiplier `MUL (0,1)` in the circuit presented in Figure 7-7. This multiplier block is initially configured to be in `(x,m,m)` or `(x,h,h)` mode. In this example, the target multiplier block has been calibrated with the `fast` calibration strategy offered by the SIMPL analog hardware. The `LScale` compilation procedure compensates for the subset of unexpected gains present in the calibrated multiplier block that are amenable to correction. These unexpected gains are provided to the compiler in the form of a delta model. Refer to Section 7.2 for an overview of the delta models associated with the multiplier block instances on the SIMPL device. The delta models for the multiplier `(0,1)` are presented below:

```
(x,m,m)   0.94875*c*x        (x,h,h)   0.87330*c*x
(x,h,m)   0.1*0.93110*c*x    (m,m,m)   0.5*1.0*x*y
```

The Master Expression

The master expression models all unexpected gains that can be corrected through circuit scaling and captures the effect of changing the block mode on the computation. This master expression contains block ports, data fields, and mode-dependent property variables called process variation property variables. The master expression for the multiplier `(0,1)` is presented below:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot c \cdot x$$

The compiler defines the proc-var(MUL (0,1),z,0) mode-dependent process variation property variable. Each process variation property variable captures some observed deviation from the reference expression under a particular mode. Each process variation property variable resolves to a positive constant coefficient when the block mode variable is fixed to an integer value.

Reference Expression: The master expression captures all deviations in behavior relative to a reference expression that describes the idealized behavior of the target output port in the unscaled circuit. The reference expression is the expression implemented at the target output port under the modes defined in the input unscaled ADP. In the running example, the input unscaled ADP configures the multiplier (0,1) to be in either (x,m,m) or (x,h,h) mode. Both modes implement the same input-output relation at port z. The reference expression for port z of multiplier (0,1) is therefore c*x. The master expression implements the reference expression when all process variation property variables are set to one:

$$1.0 \cdot c \cdot x$$

The above equation validates that the master expression for MUL (0,1) implements the reference expression c*x when the process variation property variables are set to one. This property ensures that the master expression implements the idealized dynamics of the block, provided the process variation property variables can be factored out or eliminated.

Delta Models: The compiler ensures the master expression implements the delta models associated with the block instance. The master expression elicitation procedure produces a set of CGP constraints which ensure the process variation property variables are assigned to the correct values under each mode. The compiler generates the following process variation property variable constraints for the MUL(0,1) block:

$$\texttt{mode}(\texttt{MUL(0,1)}) = \texttt{(x,m,m)} \Rightarrow \textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) = 0.94875$$
$$\texttt{mode}(\texttt{MUL(0,1)}) = \texttt{(x,h,m)} \Rightarrow \textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) = 0.093110$$
$$\texttt{mode}(\texttt{MUL(0,1)}) = \texttt{(x,h,h)} \Rightarrow \textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) = 0.87330$$

The following constraints map the process variation property variable to the appropriate value depending on the mode. Under these constraints, the master expression implements the 0.94875*c*x, 0.87330*c*x, 0.1*0.93110*c*x delta models when the mode variable mode(MUL (0,1)) for multiplier MUL(0,1) is set to the (x,m,m) (x,h,h), (x,h,m) mode respectively.

The master expression and the above implication statements together model how the delta models deviate from the reference expression. For example, setting the mode variable for multiplier MUL (0,1) to (x,m,m) specializes the master expression so that it implements 0.94875*c*x – the delta model for port z of the MUL (0,1) under (x,m,m) mode. Some of the mode variable assignments

incorporate both the delta model parameter and the effect of changing the mode. For example, the mode variable is assigned to `0.1*0.93110=0.093110` when the mode is changed to `(x,h,m)`. The delta model parameter `0.93110` is scaled by `0.1` to account for the fact that the output signal is scaled by `0.1` when the block mode is changed from `(x,m,m)` to `(x,h,m)`.

The master expression derivation procedure fails to incorporate the delta model for mode `(m,m,m)` into the master expression. The `LScale` pass therefore adds the following constraint that the mode `(m,m,m)` is not selected for the `MUL (0,1)` block:

$$\texttt{mode}(\texttt{MUL(0,1)}) \neq \texttt{(m,m,m)}$$

The compiler also produces a constraint that prevents the multiplier block `MUL(0,1)` from being put in `(m,m,m)` mode. The compiler introduces this mode restriction because it could not identify a master expression that could implement both the delta model for the block in `(m,m,m)` mode (`0.5*1*x*y`) and the reference expression (`c*x`).

Master Expression Elicitation

The master expression elicitation procedure derives the master expression and associated CGP constraints for a target output port instance from the input ADP, ADS, delta model database, and calibration strategy. The identified master expression models the effect of changing the mode on the signal and captures the effect of any unexpected gains on the signal dynamics. The master expression elicitation procedure returns the computed master expression and a set of CGP constraints which must hold for the master expression to faithfully implement the delta models associated with the target output port instance.

The master expression elicitation procedure first identifies the reference expression and the delta models for the target output port instance. The procedure derives the reference expression by looking up the input-output relation in the ADS under the modes listed in the unscaled input ADP.

Next, the master expression elicitation procedure constructs the delta models for the target output port from the ADS delta model specifications, the delta model database, and the calibration strategies. The compiler derives these delta models by substituting all correctable gain delta model parameters with the appropriate values from the delta model database and setting all other delta model parameters to their ideal values.

Finally, the master expression elicitation procedure harmonizes the delta model expressions and reference expression to produce a master expression which can be specialized to implement both the delta models and the reference expression. The harmonization procedure also produces CGP constraints which map process variation property variables to values, and CGP constraints which prevent block modes that are not captured by the master expression from being selected. Refer to

Section 9.4.2 for details on how the master expression is derived.

Expression Factoring Algorithm

The expression factoring algorithm factors out a scale expression from the scaled dynamics of the signal at the target output port. This algorithm accepts as input a master expression that describes the behavior of the target output port. It produces, as output, the scale expression which has been factored out of the scaled dynamics and a set of geometric programming constraints which must hold for the factoring procedure to complete successfully. The factored scale expression contains scale transform variables, such as magnitude and time scaling variables, injection variables, and process variation property variables that capture the device's unexpected gains. The derived factoring constraints ensure that the scaled dynamics of the signal for the device on hand equals the idealized dynamics of the signal times the factored out scale expression.

Scaled Master Expression: The expression factoring algorithm first derives a scaled master expression that captures the dynamics of the scaled signal for the device on hand. The scaled master expression models the unexpected gains present in the hardware, the effects of mode selection on the computation, and the scaling transform. The compiler obtains the scaled master expression by symbolically applying the scaling transform to the provided master expression. To symbolically apply the scale transform, the compiler multiplies all ports and data fields by their respective magnitude scale factors. I present the scaled master expression for the `MUL (0,1)` multiplier block below:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot (\texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{c}) \cdot (\texttt{mag}(\texttt{MUL(0,1)}, \texttt{x}) \cdot \texttt{x})$$

The above expression models how the signal at port `z` of multiplier block `(0,1)` would behave if the scaling transform were applied to the unscaled ADP.

Constraint Generation: The compiler next produces factor constraints which ensure an expression made up of scale transform variables and process variation property variables can be factored out of the dynamics of the scaled signal. Provided the returned factor constraints hold, the scaled dynamics of the signal should equal the factored scale expression times the reference expression. Refer to Section 9.4 for discussion on how the compiler generally derives the factor constraints and scale expression from the scaled master expression.

For the multiplier block `MUL (0,1)`, the expression factoring algorithm is able to factor out a scale expression from the scaled master expression by shuffling around the terms in the expression:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x}) \cdot (\texttt{c} \cdot \texttt{x})$$

The above factored expression is the product of the original reference expression `c*x` and a scale expression $\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x})$ made up of magnitude scale

factor and process variation terms. Note that no factor constraints are necessary to perform this factoring operation since multiplication is associative. The expression factoring algorithm returns an empty set of factor constraints and the following factored scale expression:

$$\textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x})$$

Finalization

The factor constraint generation procedure collates together the CGP constraints computed by the master expression elicitation procedure and the GP constraints computed by the expression factoring algorithm to produce a complete set of factor constraints. The factor constraint generation procedure completes the set of constraints by adding a geometric programming constraint that links the factored scale expression to the magnitude scale factor of the target output port:

$$\texttt{mag}(\texttt{MUL(0,1)}, \texttt{z}) = \textit{proc-var}(\texttt{MUL(0,1)}, \texttt{z}, 0) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{c}) \cdot \texttt{mag}(\texttt{MUL(0,1)}, \texttt{x})$$

The above constraint ensures the magnitude scale factor for the output port `z` of multiplier block `MUL (0,1)` equals the scale expression factored out of the scaled signal dynamics. The factor constraint generation procedure then returns the complete set of CGP and GP constraints and the factored scale expression.

7.6 Conclusion

This chapter presented a high-level overview of how the compiler compiles a target dynamical system to a differential equation-solving reconfigurable analog device. Throughout this chapter, I used a running example in which I mapped a simple harmonic oscillator to the SIMPL reconfigurable analog device. This the SIMPL device is a simplified dynamical system-solving analog device inspired from the HCDCv2.

The compiler first synthesizes an unscaled ADP which implements the dynamical system (circuit synthesis) on the target analog device. The compiler then scales the unscaled ADP so that it respects the physical constraints imposed by the hardware while also preserving the original dynamics of the dynamical system (circuit scaling). The scaled ADP can then be executed on the target hardware platform.

Circuit Synthesis: I first presented the operation of the circuit synthesis procedure. The circuit synthesis pass accepts as input a dynamical system specification and an analog device specification and produces an unscaled ADP.

The circuit synthesis pass first synthesizes a collection of circuit fragments that implement the relations in the dynamical system specification (synthesis procedure). The fragment synthesis procedure employs a tableau-based search algorithm to construct a circuit fragment for each dynamical system relation. This algorithm leverages an algebraic rewrite system to identify creative usages of the analog blocks. This algorithm enables the compiler to more effectively map the dynamical system relations to the analog hardware.

The circuit synthesis pass then links the fragments together to form a completed circuit (assembly procedure). The assembly procedure introduces assembly blocks as necessary to route and copy signals. Examples of assembly blocks include current copiers and signal converters. The assembly procedure sometimes needs to add additional blocks because some signals, such as analog currents, cannot be used multiple times.

The circuit synthesis pass then maps blocks to locations on the analog device, inserting route blocks when necessary (place+route procedure). Analog devices often provide dedicated route blocks which forward signals over long distances. The place+route procedure inserts these route blocks when necessary to implement the connections in the circuit.

The place+route procedure encodes the block placement problem as an integer linear programming problem. This place+route algorithm leverages the device layout information from the ADS and the insight that fewer long-distance connections are available on a given piece of hardware to decompose the problem into a sequence of smaller sub-problems. Instead of solving a single integer linear programming problem, the place+route procedure breaks up the placement problem into several smaller integer linear programming problems that incrementally resolve the location of each block to finer-grain spatial structures.

Circuit Scaling: I next provided an overview of the circuit scaling pass. The circuit scaling pass accepts as input the unscaled ADP produced by the circuit synthesis pass, the ADS and DSS, the calibration strategy to target, a scaling objective function, and the delta model database for the device on-hand. The ADS, calibration strategy, and delta model database together fully define the physical constraints and behaviors present in the device on hand. The DSS provides the variable intervals for the unscaled ADP and the scaling objective function describes what criteria to minimize when scaling the circuit.

The circuit scaling pass frames the scaling problem as a combinatorial geometric programming problem (CGP). The formulation enables the circuit scaling pass to simultaneously reprogram blocks in the unscaled ADP and derive a scaling transform. This reprogramming feature enables the scaling pass to more flexibly scale the circuit. The combinatorial geometric programming problem captures the operating range and frequency restrictions imposed by the device, encodes the effect of analog noise and quantization error on the signals, and models the empirically observed behavioral deviations present in the device on hand. The CGP also adds the necessary constraints to ensure

that the scaled ADP preserves the original dynamics of the dynamical system.

Further Reading: The compilation overview makes use of the dynamical system specification language introduced in Chapter 3 and the analog device specification and analog device programming languages introduced in Chapter 5. Refer to Chapter 6 for more discussion on unscaled and scaled ADPs. Refer to Chapter 8 for a rigorous description of the circuit synthesis pass. Refer to Chapter 9 for a rigorous description of the circuit scaling pass.

Chapter 8

Circuit Synthesis

To implement a dynamical system on an analog device, the analog building blocks resident on the device must be configured and routed together so that the physics of the device (behavior of the currents/voltages over time) matches the behavior of the dynamical system. The compiler synthesizes analog device programs that are guaranteed to be algebraically equivalent to the starting dynamical system, provided the analog blocks behave ideally and are not subject to any low-level physical behaviors. That is, each produced ADP implements a dynamical system that can be rewritten to match the original dynamical system by successively applying algebraic rewrite rules. The analog device programs produced by the circuit synthesis pass are the unscaled analog device programs presented in Chapter 6.

The analog configuration synthesis procedure accepts as input a specification of the dynamical system to compile (DSS) and a specification of the analog device (ADS). The analog device specification describes each block's behavior and programming interface and the programmable connections available on the device. It produces as output an analog device program (ADP) which implements the provided dynamical system on the target analog hardware. Internally, the compiler works with an extended representation of the analog device program (vADP) that supports defining circuit fragments. This chapter covers the following configuration synthesis steps:

Fragment Synthesis (Section 8.2): I describe how the compiler synthesizes circuit fragments which implement dynamical system relations. This fragment synthesis procedure is non-trivial because the compiler may need to compose together blocks in non-trivial ways to obtain the desired dynamics. I provide a formalization of the tableau-based synthesis procedure which produces correct-by-construction circuit fragments that are algebraically equivalent to the starting circuit. This step of compilation produces a collection of vADP fragments made up of compute blocks that implement each relation in the DSS.

Assembly (Section 8.3): I describe how the compiler assembles a complete circuit from the

synthesized circuit fragments. This assembly procedure is non-trivial as the compiler may need to copy and transform signals to connect the synthesized fragments together. The compiler, therefore, generates bridge circuits composed of assemble blocks that link fragments together. This section describes how the compiler generates these bridging circuits and incorporates these bridge circuits into the larger circuit. This step of compilation produces a complete vADP circuit from the vADP fragments generated by the synthesis procedure.

Routing (Section 8.4): I describe the compiler's place and route procedure. The place and route procedure maps the blocks and connections in the completed circuit to locations and digitally programmable interconnects on the analog device. This routing procedure is non-trivial as the compiler may need to inject route blocks to form certain connections. The routing procedure assigns each vADP block instance to a block location on the device, inserting routing blocks when necessary. It produces an analog device program (ADP) from the complete vADP circuit produced in the previous step. The resulting ADP respects any resource and connectivity constraints present in the device.

8.1 Problem Definition

`LGraph` accepts, as input, a dynamical system and analog device specification (ADS,DSS) and produces, as output, a program for the specified analog device (ADP) which implements the provided dynamical system. I introduce the notation and mathematical constructs in the following section.

8.1.1 Notation

In this chapter, I express spaces of elements with descriptive words, sets of elements belonging to that space with capital letters, and elements belonging to the space with lower case letters. For example, I might introduce a space of generic variables $GenericVars$ where a generic variable gv is in the space $GenericVars$. A set of generic variables is written as GV.

A `function` maps one space of elements to another. For example, I might introduce a `dup2` function which maps generic variables to pairs of generic variables. I formally define this function as $\texttt{dup2} : GenericVars \rightarrow GenericVars \times GenericVars$. This formalism also uses `literals` which map to constant values.

Math Primitives: This chapter makes use of several common numerical spaces and mathematical operators. The $\mathbb{N}$ and $\mathbb{R}$ quantities correspond to the space of natural numbers and the space of real numbers, respectively. I use the notation $\mathbb{R}^m$ to represent the space of real vectors of length m. The $\times$ operator computes the cartesian product of two sets or spaces, and the $\mathbb{P}$ operator computes the powerset of a set or space. This notation also makes ample use of sets, multisets, and sequences.

I denote a set with curly braces ({}). Sequences of variables are denoted with the [[]] operator. The *singleton*() operator returns the element contained by a set if the set contains a single value. Multisets are denoted with the M operator.

Nondeterminism: This chapter presents algorithms which make non-deterministic choices. The `choose`$(\{..\}, n)$ function non-deterministically selects up to n elements from a set without replacement. If the set contains less than n elements, then the `choose` function fails.

Multisets: A multiset is an ordered collection of elements that may contain multiple instances of an element. A multiset over some set of elements GV is a mapping from GV to $\mathbb{N}$ ($M : GenericVars \rightarrow \mathbb{N}$). The multiset can also be represented as a set of element-count pairs:

$$\{(gv, Mgv) \mid gv \in GenericVars\}$$

The above representation of a multiset defines the multiset as a set of element-cardinality pairs for each element gv in the space of elements $GenericVars$. Alternatively, multisets can be written as polynomials over set elements:

$$gv_0^{n_0} gv_1^{n_1} ... gv_m^{n_n} = \{(gv_0, n_0), (gv_1, n_1), ..., (gv_m, n_n)\}$$

The elements $gv_0, gv_1.., , gv_m$ are elements belonging to the space of elements $GenericVars$ and the exponents $n_0, n_1, ..., n_m$ are the counts of the listed elements. The equivalent set representation is written on the right-hand side of the equality sign. The empty multiset 0 is a multiset where all elements are mapped to zero. The following operators are standard operators for multisets unless otherwise indicated:

Multiset Inclusion ($\subseteq$): The multiset inclusion operator $\subseteq$ tests if one multiset is contained within another:

$$M \subseteq M' \; iff \;\; \forall gv \in GV, Mgv \leq M'gv$$

The inclusion operator returns *true* if the cardinality of each element in the first multiset is less than or equal to the cardinality of the element in the second multiset.

Multiset Union ($\cup$): The multiset union operator creates a new multiset which contains all the elements in the provided multisets M and M':

$$\forall gv \in GenericVars, (M \cup M')gv = (Mgv) + (M'gv)$$

The union operator crafts a new multiset $M \cup M'$ which maps gv to the sum of the individual gv counts $Mgv + M'gv$.

Multiset Difference ($-$): The multiset difference operator subtracts one multiset M from another multiset M'. The difference operator requires $M' \subseteq M$ to succeed:

$$\forall gv \in GenericVars, (M - M')gv = (Mgv) - (M'gv), if M \subseteq M'$$

Multiset Support $Supp$: The multiset support operator retrieves the set of elements that have more than one element in the provided multiset:

$$Supp\ M = \{gv \in GenericVars \mid Mgv > 0\}$$

The multiset support operator is also used to test element membership. The $gv \in Supp\ M$ clause determines if an element belongs to a multiset.

Multiset Intersection $\cap$: I introduce a new multiset intersection operator which computes a minimal multiset from two multisets M and M':

$$\forall gv \in GenericVars, (M \cap M')gv = min(Mgv, Mgv)$$

The intersection operator creates a multiset which only contains the common elements between the two provided multisets. Each reported element count is the minimum of the element counts from the provided sets.

Multisets from List (`multi`): I introduce a new operator (`multi`) which transforms a sequence of elements into a multiset. The count for each element in the multiset corresponds to the number of occurrences of that element in the provided sequence.

Multiset to Element (`element`): I introduce a new operator which transforms a multiset to a single element:

$$\texttt{element}(M) = gv\ if\ Supp\ M = \{gv\}$$

The singleton operator requires that the support of the multiset contains exactly one element. It returns the element contained in the multiset.

Size of Multiset: I introduce a new operator which computes the size of a multiset:

$$\texttt{size}(M) = \sum_{gv \in GenericVars} Mgv$$

The size operator returns the sum of all the cardinalities of all the elements in the multiset.

Expressions: This formalism works with symbolic mathematical expressions $e \in Exprs$ over variables $v \in Vars$. The $\texttt{vars} : Exprs \to \mathbb{P}(Vars)$ function returns the variables which appear in the expression. The $e[v/e']$ notation replaces all of occurrences of the variable v in e with e'. I generally express a variable-expression assignment with the assignment construct $a \in Assignments = Vars \times Exprs$. The $\texttt{sub} : Exprs \times \mathbb{P}(Assignments) \to Exprs$ function accepts an input expression

and a set of assignments and returns the expression with the all the variable-expression substitutions applied. For example, the $\texttt{sub}(v+v', \{(v, e), (v', e')\})$ invocation of the substitution function returns the expression $e + e'$.

Note that these expressions may contain dynamical system variables, hardware variables, or a mixture of the two. Therefore, the space of variables $Vars$ contains variables that appear in the hardware specification and the dynamical system specification.

8.1.2 Dynamical System Specification

The dynamical system specification is comprised of dynamical system variables $dv \in DSVars$, where the dynamics of each variable dv is modeled as expression $de \in DSExprs = \{e \in Exprs \mid \texttt{vars}(e) \subseteq DSVars\}$. The $\texttt{dsexpr} : DSVars \rightarrow DSExprs$ function maps dynamical system variables to dynamical system expressions.

8.1.3 Analog Device Specification

`LGraph` works with a collection of programmable compute blocks $cb \in ComputeBlocks$, assembly blocks $xb \in AssembleBlocks$, and routing blocks $rb \in RouteBlocks$. Together, these blocks make up the blocks $b \in blocks$ available on the analog hardware.

Blocks have input ports $ip \in InputPorts$ and output ports $op \in OutputPorts$. Each port works with a signal of type $st \in SignalType$ where $SignalType = \{\texttt{digital}, \texttt{current}, \texttt{voltage}\}$. The $\texttt{sigtype} : InputPorts \bigcup OutputPorts \rightarrow SignalType$ function maps ports to signal types. Each block provides a digital programming interface to the compiler in the form of a digitally settable mode $m \in Modes$ and collection of data fields $df \in DataFields$. Each data field is associated with a type $dft \in DFT$ where $DFT = \{\texttt{constant}, \texttt{expression}\}$. Constant data fields (`constant`) map to constant values $y \in \mathbb{R}$. Expression data fields (`expression`) map to symbolic expressions.

The $\texttt{df-type} : DataFields \rightarrow DFT$ function maps data fields to data field types. The $\texttt{block} : DataFields \bigcup InputPorts \bigcup OutputPorts \rightarrow Blocks$ function returns the block that contains the specified input port, output port, or data field.

Each block output port $op \in OutputPorts$ implements a collection of expressions $he \in HwExprs$, where $HwExprs = \{he \in HwExprs \mid \texttt{vars}(he) \subseteq InputPorts \cup DataFields\}$. The exact hardware expression implemented at each output port depends on the mode of the block. The $\texttt{portexpr} : OutputPorts \times Modes \rightarrow HwExprs$ function returns the expression implemented by each output port for each block mode.

The hardware specification language enables the designer to specify multiple instances of the declared blocks in the device specification. Each *block instance* $bi \in BlockInst = Blocks \times Locs$ where $bi = (b, l)$ is identified as a block $b \in blocks$ at location $l \in Locs$. In this chapter, I write

block instances as block-location tuples (b, l). The specification defines the available programmable connections between ports of block instances $c \in Conns = InputPorts \times Locs \times OutputPorts \times Locs$.

The hardware specification works with a description of the spatial layout of the device. Each of the device is organized into $n \in \mathbb{N}$ sequentially organized spatial views numbered $1...i...n$. Each spatial view i contains one or more spatial locations $sl_i \in SpatialLocs_i = \mathbb{N}^{i+1}$. Every location in a particular view also belongs to a location in all preceding views. Block instances may only be bound to locations from the most specific view (the leaf view). The space of block instance locations corresponds to the set of locations belonging to the most specific view $Locs = SpatialLocs_n$. I use the `spat-member`$(sl_{sv}, sl'_{sv'})$ function to test if some location sl_{sv} is contained by spatial location $sl'_{sv'}$.

8.1.4 Analog Device Program

The solver generates an analog device program (ADP) which implements an analog circuit on the target analog hardware. The ADP is made up of a collection of statements $z \in Stmts$ which describe the subset of programmable connections to enable, how to configure each block, and which signals correspond to which dynamical system variables:

- `conn`(op, l, ip, l'): This statement enables the connection from output port op of block instance (`block`$(op), l)$ to input port ip of block instance (`block`$(ip), l')$
- `config`(b, l, M, X): This statement configures the block b at location l. The configuration $X \subseteq Configs = DataFields \times DSExprs$ describes how to set each data field in the block and the mode set M specifies the set of modes that deliver the desired behavior for that block instance.
- `source`(op, l, dv): This statement maps the signal of type `sigtype`(op) at output port op the the dynamical system variable dv. The evolution of this signal `sigtype`(op) at this port is analogous to the evolution of the variable dv in the dynamical system.

8.1.5 Virtual Analog Device Program

The vADP is made up of a collection of statements $vz \in VStmts$ that together describe an analog circuit or analog circuit fragment. In contrast with ADPs, vADPs supports the specification of circuit fragments and identify block instances with abstract numerical identifiers $i \in \mathbb{N}$ instead of device locations $l \in Locs$. The vADP distinguishes between multiple uses of the same block using a block identifier $i \in \mathbb{N}$. Each virtual block instance $vbi \in VirtBlockInst = Blocks \times \mathbb{N}$ is later mapped to a physical block instance $bi \in BlockInst$ on the analog device. In this chapter, I write virtual block instances as block-identifier tuples (b, i). The vADP accepts the following statements

- `vconn`(op, i, ip, i'): This statement connects the output port op of block `block`(op) with identifier i to input port ip of block `block`(ip) with identifier i'.
- `vconfig`(b, i, M, x): This statement configures the block b with identifier i. The block modes are set to M and the programmable block fields are configured using configuration $x \in Configs$.
- `vsink`(ip, i, de): This statement describes where an incoming signal is needed in the vADP circuit. A signal of type `sigtype`(ip) that implements dynamical system expression de must be provided to input port ip of block `block`(ip) with identifier i to complete the circuit. A vADP is a *vadp fragment* if vADP contains any `vsink` statements.
- `vsource`(op, i, de): This statement maps the signal at an output port op of block `block`(op) with identifier i to a dynamical system expression de. The evolution of the signal of type `sigtype`(op) at that port matches the trajectory of de over time.

8.2 vADP Fragment Synthesis

The vADP fragment synthesis algorithm accepts, as input, the dynamical system specification with dynamical system variables DV and a a hardware specification with compute blocks CB which contain input ports IP, and output ports OP. For each variable $dv \in DV$ it produces a collection of vADP fragments. Each vADP implements the dynamics of dv (`dsexpr`$(dv) = de$) using a circuit fragment made up of compute blocks.

8.2.1 The Tableau

The synthesis algorithm works with algebraic structure called a *tableau*. The tableau $t \in Tableaus = \mathbb{P}(Goals) \times \mathbb{P}(Relations) \times \mathbb{P}(VStmts)$ is composed of goals $g \in G \subseteq Goals$ to solve, a set of hardware relations $r \in R \subseteq Relations$ are used to solve these goals, and a set of vADP statements $VZ \subseteq VStmts$ that describe the vADP fragment.

Goals: The tableau goals describe the dynamics which still need to be implemented in the vADP fragment. Each goal $g \in Goals$ or `goal`$(ip, i, e))$ describes a mathematical relation that must be implemented on the analog hardware. A goal of the form `goal`(ip, i, de) assigns the signal fed into input port ip to a dynamical system expression. A goal of the form `goal`(dv, de) maps the dynamical system variable dv to a dynamical system expression de. The expression de is referred to as the dynamics of the goal or the goal expression.

Relations: The tableau relations describe the available compute operations on the analog device. Each relation describes a mathematical expression that may be implemented on the analog hardware.

Each relation $\texttt{rel}(op, i, M, he)$ describes the dynamics (he) implemented by port op belonging to block $\texttt{block}(op)$ with identifier i when one of the modes $m \in M$ is selected. I refer to the hardware expression he as the dynamics of the relation.

vADP Statements: The tableau vADP statements encode the circuit fragment produced by this procedure. The synthesis algorithm continually extends the circuit encoded with the vADPs to implement more tableau goals.

8.2.2 Basic Approach

The synthesis algorithm accepts as input a dynamical system variable to synthesize (dv) and a dynamical system and analog device specification. It produces a virtual analog device program that implements the dynamical system variable's dynamics on the analog hardware.

The compiler first constructs an *initial tableau* t_0 which contains a single goal describing the dynamics of the target variable. It then nondeterministically applies a transition relation $\rightarrow$ to the initial tableau to derive more tableaus:

$$\texttt{synth}(dv) = \{VZ \mid \langle \texttt{goal}(dv, \texttt{dsexpr}(dv)), R_0, \emptyset \rangle \rightarrow^* \langle \emptyset, R, VZ \rangle\}$$

It performs this procedure until it finds a *solved tableau*. A tableau is marked as solved when there are no more goals left. The set of vADP statements in the solved tableau encodes a circuit fragment that implements the dynamics of the target dynamical system variable dv.

8.2.3 The Initial Tableau

Given a dynamical system variable dv to synthesize, the compiler constructs a initial tableau t_0:

$$t_0 = \langle \{\texttt{goal}(dv, \texttt{dsexpr}(dv))\}, R_0, \emptyset \rangle$$

The initial tableau has a single goal $\texttt{goal}(dv, \texttt{dsexpr}(dv))$ which encodes the dynamics of dv. To eliminate this goal, the synthesis algorithm must synthesize a circuit that implements the expression $\texttt{dsexpr}(dv)$. The initial tableau also contains a set of starting relations (R_0) and an empty vADP. The starting relation set contains a relation for each distinct expression he implemented by each output port $op \in OP$:

$$R_0 = \{\texttt{rel}(op, 0, M, he) \mid \texttt{portexpr}(op, m) = he \; \forall \; m \in M \wedge op \in OP\}$$

The mode set M specifies the set of modes $m \in M$ that, when selected, implement hardware expression he at port op. All the relations in the starting relation set together capture the range of

functions the analog hardware can implement.

8.2.4 The Solved Tableau

A tableau t_{sln} is considered solved if there are no goals left. The set of vADP statements in the tableau is the vADP fragment which implements the starting dynamical system variable.

$$t_{sln} = \langle \{\}, R, VZ \rangle$$

8.2.5 The $\rightarrow$ Operator

The synthesis algorithm (`synth`) produces a set of solved tableaus from the initial tableau by repeatedly applying the tableau transition operator $\rightarrow$.

$$\texttt{synth}(dv, de) = \{VZ \mid \langle \texttt{goal}(dv, de), R_0, \emptyset \rangle \rightarrow^* \langle \emptyset, R, VZ \rangle\}$$

The synthesis algorithm accepts as input the dynamical system variable (dv) and its dynamics (de). It produces a set of solution tableaus that implement the dynamical system variable on the analog device. I formalize the operation of the synthesis algorithm as a transition relation $t \rightarrow t'$, where each transition corresponds to a solver step on the tableau.

1. **Goal and Relation Selection.** The tableau transition operation first selects a goal g and a relation r which describes the dynamics of some output port *op* under some set of modes M.
2. **Unification**: The transition then derives a set of assignments that concretizes the dynamics of the signal emitted at output port *op* such that it is algebraically equivalent to the dynamics of the goal. Two expressions e and e' are *algebraically equivalent* if there exists a set of algebraic rewrite rules which transforms e into e'.
3. **Application**. It applies these assignments to the tableau to derive an updated tableau. The application process resolves the target goal, consumes the target relation, and extends the vADP to implement the goal dynamics with the selected relation.

8.2.6 Goal and Relation Selection

The transition relation $\rightarrow$ is a one-way transition that solves a single goal in the provided tableau. First, the transition relation selects a goal g and relation r. It then checks the goal and relation to make sure they work with the same type of signal. This type-checking operation ensures that only ports which work with the same type of signal are connected together in the produced circuit.

$$\frac{\texttt{typecheck}(g, r) \quad \ldots}{\langle \{g\} \cup G, \{r\} \cup R, VZ \rangle \rightarrow \ldots}$$

The `typecheck` operation accepts a goal and relation and determines if the goal and relation work with the same type of signal. The behavior of the typechecking operation is described below:

function TYPECHECK(g,r)
  match $\langle r, g \rangle$ **with**
    | $\langle \texttt{rel}(op, i, M, he), \texttt{goal}(dv, de) \rangle$ -> `true`
    | $\langle \texttt{rel}(op, i, M, he), \texttt{goal}(ip, i', he') \rangle$ -> $\texttt{sigtype}(ip) = \texttt{sigtype}(op)$

The `typecheck` operation always passes if the selected goal maps a dynamical system variable to a dynamical system expression. The compiler can use any available hardware relation to satisfy such a goal. If the goal maps an input port to a dynamical system expression, the signal type of the input port must match the signal type of the output port chosen relation:

8.2.7 Unification

The transition relation then unifies the dynamics of the selected hardware relation r with the dynamics of the selected goal g:

$$\frac{\texttt{typecheck}(g, r) \quad \texttt{unify}(g, r) = A \quad \ldots}{\langle \{g\} \cup G, \{r\} \cup R, VZ \rangle \rightarrow \ldots}$$

The unification (`unify`) procedure accepts as input a relation $r = \texttt{rel}(op, i, M, he)$ and a goal of the form $\texttt{goal}(dv, de)$ or $\texttt{goal}(ip, i, de)$ and specializes the enclosing block $\texttt{block}(op)$ to fulfill the provided goal. It computes a set of assignments A which specialize the block to implement the dynamical system expression de at output port op. The unification procedure uses the expression unification (`unify-expr`) algorithm to produce the set of assignments from the hardware expression and dynamical system expression.

The $\texttt{unify-expr}(he, de)$ function accepts a hardware expression and dynamical system expression as input and produces a set of assignments. These assignments map block data fields or input ports in the virtual block instance ($vbi = (\texttt{block}(op), i)$) to dynamical system expressions. The `unify-expr` routine guarantees the dynamics of hardware expression he with the assignments applied is *algebraically equivalent* to the dynamical system expression de. Two expressions are algebraically equivalent if they implement the same input-output relation:

$$de \equiv \texttt{sub}(he, A)$$

The signal at output port *op* implements *de* when the data field assignments in A are applied to the virtual block instance, and each input port is supplied with a signal that implements the assigned expression. A virtual block instance is considered *specialized* once it has been partially or fully configured to implement a set of expressions at its output ports.

Expression Unification (`unify-expr`)

Expression unification procedure ($\texttt{unify-expr}(he, de)$) identifies a set of port- and data field expression assignments which render the hardware and dynamical system expressions algebraically equivalent.

The unification procedure treats each port and data field in the hardware expression as a hole ($\square$) that can be filled with a dynamical system expression *de*. The procedure may optionally accept constraints that limit what types of dynamical system expressions can be bound to certain holes. The expression unification procedure applies the following constraints to the expression unification procedure:

- **Constant Data Fields**: Data fields *df* which accept constant values $\texttt{df-type}(df) = \texttt{constant}$ can only be map to real numbers $y \in \mathbb{R}$.

The expression unification algorithm uses an algebraic unification engine to unify the hardware and dynamical system expressions. The algebraic unification algorithm progressively applies algebraic rewrite rules to the target hardware expression. It produces a set of port- and data field-assignments that can be substituted into the hardware expression to produce a concrete expression over dynamical system variables. The resulting concretized expression is algebraically equivalent to the target dynamical system expression. This unification technique is well suited for analog devices which expose complex analog blocks that implement highly nontrivial algebraic functions. For example, an algebraic unification approach would be able to compute a set of assignments that unify the dynamical system expression dv^{-4} with the following hardware relation:

$$\frac{ip_1}{\left(\frac{ip_2}{ip_3} + 1\right)^{df}}$$

The above hardware relation is algebraically equivalent to the dynamical system expression when *df* is 4, ip_1 is 1, ip_2 is $(dv - 1)$, and ip_3 is 1. I can substitute the following assignments into the above hardware expression and derive the expression dv^{-4} by applying algebraic rewrite rules:

$$\frac{1}{\left(\frac{dv-1}{1} + 1\right)^4} = dv^{-4}$$

8.2.8 Applying the Unification to the Tableau

After the algorithm unifies the goal g and relation r, `LGraph` applies the unification assignments A to the starting tableau to derive a new tableau that solves the goal g with the relation r:

$$\frac{\begin{array}{c}\texttt{typecheck}(g, r) \qquad \texttt{unify}(g, r) = A \\ r = \texttt{rel}(op, i, M, he) \qquad G' = \{\texttt{goal}(ip, i, de) \mid \langle ip, de\rangle \in A\} \\ R' = \texttt{apply-rel}(r, A, R) \\ VZ' = \texttt{apply-vadp}(g, r, A, VZ)\end{array}}{\langle \{g\} \cup G, \{r\} \cup R, VZ\rangle \rightarrow \langle G \cup G', R', VZ'\rangle}$$

The transition relation extends the circuit fragment to include the unified hardware relation (VZ'), augments the set of goals to include any input port assignments imposed by the unification operation (G'), and updates the set of relations to capture the effect of the unification on the analog hardware (R'). The `apply-rels` algorithm applies the unification to the starting tableau relations, and the `apply-vadp` algorithm applies the unification to the starting vADP fragment.

The transition relation produces a new goal for each input port assignment in the set of unification assignments A. Future $\rightarrow$ executions will resolve these goals to subcircuits which provide the desired dynamical system expressions to each of these input ports.

8.2.9 Applying the Unification to the vADP

Algorithm 1 presents the algorithm for applying the unification to the tableau vADP. The `apply-vadp` routine accepts, as input, the relation $r = \texttt{rel}(op, i, M, he)$ and goal g that were used in the unification procedure, and the set of port and data field assignments A returned by the unification procedure. It returns an updated vADP VZ' which specializes virtual block instance associated with relation r to implement the unification and connects it to the rest of the circuit.

The provided unifications configure the virtual block instance $(\texttt{block}(op), i)$ containing hardware relation $r = \texttt{rel}(op, i, M, he)$. The algorithm first identifies the any existing block configuration statements in the vADP involving virtual block instance $(\texttt{block}(op), i)$. Lines 7-9 of the algorithm identify the subset of vADP statements that configure the virtual block instance containing the hardware relation r. The algorithm then identifies the subset of assignments in A that map expressions to block data fields. The compiler uses these assignments to configure the block. The derived data field assignments X and modes M together configure the virtual block instance to implement the required dynamics at output port op. If the vADP VZ already contains a configuration for the virtual block instance $(\texttt{block}(op), i)$ associated with relation r, then it updates that configuration to include the new data field assignments and modes (line 16). This updated configuration preserves the dynamics of any other previously specialized output ports in the virtual block instance. Other-

Algorithm 1 Algorithm for applying unification assignments to vADP in tableau (`apply-vadp`)

```
 1: # g: the goal to solve
 2: # r: the hardware relation to unify with the goal
 3: # A: the port/data-field assignments returned by the unification algorithm
 4: # VZ: the current vADP from the tableau
 5: # returns: the updated vADP with the unification applied
 6: function APPLY-VADP(g, r, A, VZ)
 7:     let rel(op, i, M, he) = r
 8:     let cb = block(op)
 9:     let VZ_config = {vconfig(cb', i', M', X') ∈ VZ | cb' = cb ∧ i = i'}
10:     let X = {(df, de) ∈ A}
11:     let vz = match g with
12:         | goal(dv, de) -> vsource(op, i, dv)
13:         | goal(ip, i', de) -> vconn(op, i, ip, i')
14:
15:     let vz' = match VZ_config with
16:         | {vconfig(_, _, M', X')} -> vconfig(cb, i, M ∩ M', X ∪ X')
17:         | ∅ -> vconfig(cb, i, M, X)
18:
19:     assert valid vz'
20:     return {vz, vz'} ∪ (VZ/VZ_old)
```

wise, it creates a fresh vADP configuration statement which writes the data field assignments and modes to the block instance (line 17). The algorithm also creates a vADP statement that links the circuit encoded in the tableau vADP to the newly configured block (lines 11-13).

It may not be possible to specialize a block instance that another unification operation has previously specialized. Line 19 tests the validity of the block configuration. An *invalid* `vconfig` statement either has no valid modes (M is $\emptyset$) or maps a data field to two different values ($(df, e) \in X$ and $(df, e') \in X$). Such configurations cannot be programmed to the device and are marked as invalid. If the `vconfig` statement is invalid, the compiler cannot apply the unification to the tableau, and the transition operation fails.

The algorithm then derives a vADP statement (vz') which routes the signal at output port op to where it is needed. If the target goal g maps the dynamical system expression de to the input port ip of block instance `block`(ip, i'), it produces a `vconn` statement connecting output port op to the input port ip. Note that the type-checking operation `typecheck` used earlier in the synthesis procedure ensures that op and ip work with signals of the same type. If the target goal g maps the dynamical system expression de to a dynamical system variable dv, it produces a `vsource` statement which labels the output port op with the dynamical system variable dv.

Algorithm 2 Algorithm for applying unification assignments to available hardware relations in tableau (`apply-rel`)

1: # **r**: The hardware relation used in the unification
2: # **A**: The port and data field assignments computed by the unification algorithm.
3: # **R**: The set of hardware relations from the tableau.
4: # **returns** an updated set of relations with the appropriate relations specialized.
5: **function** APPLY-REL(r, A, R)
6: **let** $\texttt{rel}(op, i, M, he) = r$
7: **let** $cb = \texttt{block}(op)$
8: **let** $R_{old} = \{\texttt{rel}(op', i', M', he') \in R \mid \texttt{block}(op') = cb \wedge i = i'\}$
9: **let** $R_{updated} = \{\texttt{rel}(op', i, M' \cap M, \texttt{sub}(he', A)) \mid \texttt{rel}(op', i', M', he') \in R_{old} \wedge M' \cap M \neq \emptyset\}$
10: **let** $i_{latest} = max(\{i' \mid \texttt{rel}(op', i', M', he') \in R \wedge cb = \texttt{block}(op')\})$
11: **if** $i_{latest} == i$ **then**
12: **let** $i_{new} = i_{latest} + 1$
13: **let** $R_{new} = \{\texttt{rel}(op', i_{new}, \texttt{modes}(op, he'), he') \mid op \in OP \wedge \texttt{block}(op) = cb\}$
14: **else**
15: **let** $R_{new} = \emptyset$
16: **let** $R_{unaffected} = R / R_{old}$
17: **return** $R_{unaffected} \cup R_{new} \cup R_{updated}$

8.2.10 Applying the Unification to Tableau Relations

Algorithm 2 presents the algorithm for applying the unification to the tableau relations. The unification application algorithm (`apply-rel`) updates the tableau relations to incorporate the unification produced by the `unify` routine. The `apply-rel`(r, A, R) routine accepts, as input, the relation which was used in the unification procedure $r = \texttt{rel}(op, i, M, he)$, the set of port and data field assignments A returned by the unification procedure and the set of relations R from the tableau. It returns an updated set of hardware relations that incorporates the unification information. The returned set of relations specializes the virtual block instance containing the provided hardware relation and adds a fresh instance of the specialized block to replenish the partially (or fully) specialized relations.

The `apply-rel` function first identifies the set of relations R_{old} that to the partially specialized virtual block instance $(\texttt{block}(op), i)$ (line 6-8). It then further specializes the affected relations by concretizing the associated hardware expression using the input port and data field assignments (A) and only allowing for modes that implement the selected relation r (M). If a relation is identified as invalid, the compiler excludes it from the set of updated relations $R_{updated}$ (line 9). A relation is considered invalid if it contains no viable modes that implement both relations.

If the op port belongs to a previously unused block (i equals i_{max}), the `apply-rel` algorithm creates a new block of the same type and constructs a set of fresh relations for that block R_{new} (lines 11-13). The algorithm first identifies if the selected relation r belongs to the most recently created virtual block instance of block cb (lines 10-11). If so, the algorithm creates a new virtual block instance (cb, i_{new}) and adds the associated relations.

The compiler also includes any relations from the original relation set that are unaffected by the

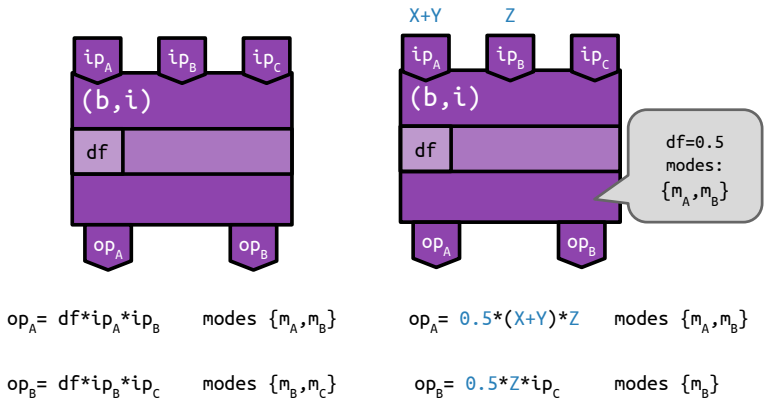


Figure 8-1: Specialization of virtual block instance with multiple outputs

unification operation $R_{unaffected}$ in the updated set of relations (line 16).

Illustrative Example

Figure 8-1 presents a a virtual block instance (b, i) with multiple output ports before and after it has been specialized by the `unify` routine. The virtual block instance has three input ports (ip_A, ip_B, ip_C), one data field df, three modes (m_A,m_B, and m_C) and two output ports (op_A,op_B). The op_A port implements $df \cdot ip_A \cdot ip_B$ when in mode m_A or m_B and op_B implements $df \cdot ip_B \cdot ip_C$ when (b, i) is in mode m_B or m_C. Prior to unification, the set of tableau relations R contains exactly two relations, both of which are associated with this virtual block instance:

$$r_A = \texttt{rel}(op_A, b, i, \{m_A, m_B\}, df \cdot ip_A \cdot ip_B)$$
$$r_B = \texttt{rel}(op_B, b, i, \{m_B, m_C\}, df \cdot ip_B \cdot ip_C)$$

Figure 8-1 presents the virtual block instance (b, i) after the relation r_A has been used to implement the expression $0.5 \cdot dv_Z \cdot (dv_X + dv_Y)$. In this block, input port ip_A implements $dv_X + dv_Y$ and input port ip_B implements dv_Z. The input port assignments are $A = \{(ip_A, dv_X + dv_Y), (ip_B, dv_Z)\}$) and the data field df is assigned to 0.5 ($X = \{(df, 0.5)\}$).

The `apply-rel`(r_A, X, A, R) function removes any relations associated with the expended output port op_A (r_A) and updates the relation r_B so that it is consistent with the partially concretized virtual block instance:

$$r_B = \texttt{rel}(op_B, b, i, \{m_B\}, 0.5 \cdot dv_Z \cdot ip_C)$$

The relation r_A places the block in mode m_A or m_B to deliver the desired expression at op_A. The algorithm updates relation r_B to only specify mode m_B – the only mode that implements both r_A and r_B. Next, the algorithm concretizes r_B's hardware expression using the input port assignments and data field assignments.

The `apply-rel` routine ensures that the returned relation set contains exactly one fresh (previously unused) virtual block instance of each block kind. It therefore adds a fresh set of relations ($r_{A'}$ and $r_{B'}$) for a new instance i' of block b after updating r_B in R:

$$r_B = \texttt{rel}(op_B, b, i, \{m_B\}, 0.5 \cdot dv_Z \cdot ip_C)$$
$$r_{A'} = \texttt{rel}(op_A, b, i', \{m_A, m_B\}, df \cdot ip_A \cdot ip_B)$$
$$r_{B'} = \texttt{rel}(op_B, b, i', \{m_B, m_C\}, df \cdot ip_B \cdot ip_C)$$

8.2.11 Putting it all Together

The inference rule presents the formalization the $\rightarrow$ unification transition below. This is the primary rule the transition operator uses to build up the vADP:

$$\frac{\begin{array}{c}\texttt{typecheck}(g, r) \qquad \texttt{unify}(g, r) = A \\ r = \texttt{rel}(op, i, M, he) \qquad G' = \{\texttt{goal}(ip, i, de) \mid \langle ip, de\rangle \in A\} \\ R' = \texttt{apply-rels}(r, A, R) \\ VZ' = \texttt{apply-vadp}(g, r, A, VZ)\end{array}}{\langle \{g\} \cup G, \{r\} \cup R, VZ\rangle \rightarrow \langle G \cup G', R', VZ'\rangle}$$

The $\rightarrow$ operator resolves trivial goals that map some input port ip to a dynamical system variable dv by inserting `vsink` statements into the vADP. This statement is later satisfied in the assembly phase with an incoming signal of the same type carrying the requested dynamical system variable.

$$\frac{\texttt{sigtype}(ip) = st \qquad vz = \texttt{vsink}(ip, i, st, dv)}{\langle g(ip, i, dv) \cup G, R, VZ\rangle \rightarrow \langle G, R, \{vz\} \cup VZ\rangle}$$

8.2.12 Computation with Physical Laws

Some analog devices perform computation by leveraging the physical behavior of a signal. For example, many current-mode devices require developers use Kirchhoff's law to add analog currents together. These devices, therefore, do not provide `compute` blocks that perform addition. Kirchhoff's law states that at any point in a circuit, the sum of incoming currents equals the sum of outgoing

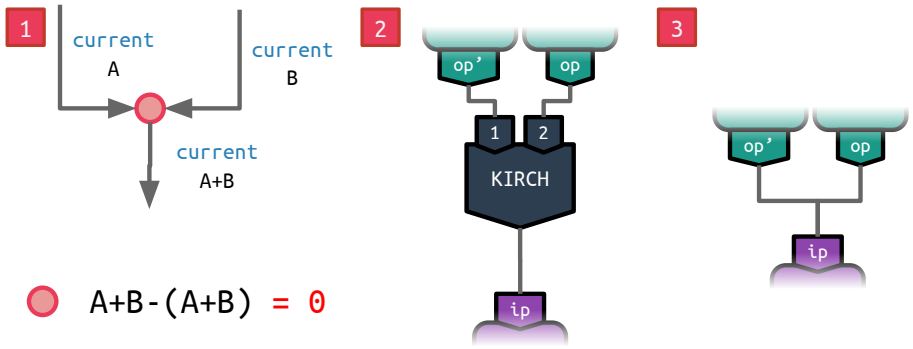


Figure 8-2: Kirchhoff's law. (1) Illustration of the law. (2) Addition circuit before simplification (3) Addition circuit after simplification

currents. Under this paradigm, two analog currents may be added together by routing them to the same input port. Figure 8-2 presents a diagram illustrating how Kirchoff's law works.

The compiler can produce circuits that leverage the physical laws governing the signal to perform computation. Each physical law is encoded as a specialized relation that describes the physics of the law. The synthesis algorithm populates the set of available relations in the tableau from both compute blocks and physical laws. In the synthesis procedure, physical law relations are treated differently from other relations in two key ways:

1. **Physical Law Application**: Each physical law may override how the unification is applied to the vADP statements in the tableau (Section 8.2.9). These application procedures may optionally introduce physical law variables ($lv \in LawVars$) into the vADP which are later resolved to virtual block instance and ports.

2. **Physical Law Simplification**: Each physical law provides a vADP simplification procedure which eliminates all physical law variables from a target vADP.

A tableau physical law relation `law`(lv_{app}, i, e) is comprised of an expression e over law variables, an identifier i that uniquely identifies the usage of the law, and a law variable lv_{app} that captures the outgoing signal for the described law. Each law variable (lv) has an associated signal type.

The synthesizer applies the physical law-specific application procedure whenever it unifies a goal with a physical law relation. It applies the simplification procedure to eliminate any law variables before returning the synthesized vADP fragment. Because physical law computations do not require dedicated hardware, they cannot be exhausted and always replenished in the tableau.

Illustrative Example: Kirchoff's law

The `LGraph` synthesizer can perform addition computations by leveraging Kirchoff's law. It implements Kirchhoff's law with the following relation:

$$\texttt{law}(lv_{kirch}, i, lv_{k,1} + lv_{k,2})$$

The above relation describes a law variable lv_{kirch} which implements the sum of two input law variables. All the law variables work with signals of type `current`. In the following example, I describe how the synthesis procedure uses Kirchhoff's law to satisfy the goal $\texttt{goal}(ip, i_{inp}, 2 * X + 4 * Y)$.

The synthesizer unifies Kirchhoff's law with the goal. This produces the subgoals $lv_{k,1} = 2 * X$ and $lv_{k,2} = 4 * Y$ and adds the following vADP statement to the vADP:

$$\texttt{vconn}(lv_{kirch}, i, ip, i_{inp})$$

The above vADP connection statement forwards the signal after Kirchhoff's law is applied to the goal's input port. A fresh relation implementing Kirchhoff's law is also added back into the tableau. The synthesis procedure continues until it identifies a solved tableau. The solved tableau's vADP will contain a set of statements involving the first invocation of Kirchhoff's law:

$$\texttt{vconn}(lv_{kirch}, i, ip, i_{inp}) \quad \texttt{vconn}(op_1, i_1, lv_{k,1}, i) \quad \texttt{vconn}(op_2, i_2, lv_{k,2}, i)$$

The above vADP fragment uses Kirchhoff's law to calculate the sum of the signals at $(op, i'$ and (op', i'') to input port (ip, i'''). All the involved law variables reference the i^{th} usage of Kirchhoff's law. `LGraph` applies the law's simplification algorithm to eliminate any law variables from the vADP:

$$\texttt{vconn}(op_1, i_1, ip, i_{inp}) \quad \texttt{vconn}(op_2, i_2, ip, i_{inp})$$

The above vADP fragment performs addition by directly connecting both outputs to the input port (ip, i_{inp}). The associated simplification algorithm effectively translates a usage of Kirchhoff's law to a circuit structure.

8.2.13 Search Algorithm

Algorithm 3 vADP fragment synthesis search algorithm

```
1: # t_0: Initial tableau containing dynamical system relation.
2: # returns the vADP fragment which implements the dynamical system relation
3: function SEARCH(t_0)
4:     T_frontier = {t_0}
5:     while T_frontier ≠ ∅ do
6:         t = {⟨G, R, VZ⟩}=choose(T_frontier, 1)
7:         if G ≠ ∅ then
8:             T = choose({t' | t → t'}, n)
9:             T_frontier = (T_frontier/{t}) ∪ T
10:        else
11:            return VZ
12:    return no solution
```

Algorithm 3 presents the search algorithm employed by the circuit synthesis procedure (`LGraph`). The algorithm explores the search space defined by the tableau transition relation $\rightarrow$. It maintains a set of tableau configurations $T_{frontier}$ at the frontier of the explored space. At each step, the algorithm chooses a tableau configuration $t : \langle G, R, VZ\rangle \in T_{frontier}$ to explore. If all of the goals in the chosen tableau $t = \langle G, R, VZ\rangle$ have been solved (i.e., $G = \emptyset$), the algorithm returns the tableau vADP fragment VZ. Otherwise, it selects a subset T of the set of tableau configurations directly reachable from t under $\rightarrow$ and replaces t in $T_{frontier}$ with T. The decisions that `LGraph` makes when it chooses a tableau configuration t (line 6) and selects a subset T of new tableaus to explore (line 8) have a significant impact on the effectiveness of the search algorithm.

Choosing t: In line 6, `LGraph` applies a heuristic designed to find the tableau configuration t that is closest to being solved. This heuristic scores each tableau configuration based on the maximum complexity of the remaining goals. The tableau selection heuristic chooses the tableau t with the smallest score.

Selecting T: `LGraph` prioritizes solving goals $g \in G$ which have fewer algebraic operators. `LGraph` therefore applies the unification algorithm to the least complex goal in the tableau. `LGraph` repeatedly applies the tableau transition rule $\rightarrow$ to tableau t to obtain T, a set of derived tableaus. If any of the goals are unsolvable, `LGraph` sets $T = \emptyset$, effectively pruning the entire search subspace rooted at t. The rationale is that the unsolvable trivial goal ensures that the pruned subspace contains no solved tableau configurations.

The $\rightarrow$ operator may generate a large number of tableau configurations. To maintain the tractability of the search algorithm, the current `LGraph` implementation discards generated tableau configurations so that T contains, at most, n tableau configurations from each explored r and g combination.

Search Depth: The depth of the search procedure is technically bounded as there are a limited number of instances of each block to use. However, in practice, it may take a long time to explore the

space. For this reason, our algorithm accepts a maximum depth. This maximum depth restriction imposes an upper limit on how many goals the search algorithm should solve before giving up.

8.2.14 Synthesis Optimizations

`LGraph` implements several extensions to the synthesis procedure that enables the formation of more sophisticated circuits. I summarize these extensions below:

Resource Limitations: `LGraph` reasons about resource limitations when synthesizing vADP assembly fragments `LGraph` does not produce relations for a fresh virtual block instance if the tableau vADP has already exhausted all the block instances available on the analog device. Similarly, `LGraph` will reject transitions that connect ports that cannot be connected together in hardware.

Short-circuit Detection: `LGraph` detects tableau vADPs that contain cycles and eliminates such tableaus from the search space. This is done because the behavior of cycles between stateless compute blocks in analog hardware is not well-defined. In some situations, forming such a cycle may introduce a short circuit and damage the hardware. The compiler allows for cycles if the cycle contains at least one stateful integrator block.

8.3 Assembly

The assembly procedure builds a complete vADP circuit from a set of vADP fragments which each implement a dynamical system variable. The *complete* vADP circuit models all the dynamical system variables in the dynamical system and has fulfilled all the `vsink` statements in the circuit. I refer to a `vsink` statement as *fulfilled* if the associated input port is connected to an output port carrying a signal of the desired type that implements the desired expression. The assembly procedure works with signals implemented dynamical system variables and expressions. A circuit signal $s \in Signals = SignalType \times DSExprs$ implements a dynamical system expression with a physical circuit property. The assembly procedure operates in three phases:

1. **Circuit Collation**: This stage collates together the vADP fragments generated by the synthesis compilation pass to form a disconnected circuit that implements all dynamical system variables in the dynamical system.

2. **Assembly Fragment Synthesis**: This stage produces a set of vADP fragments that satisfy all the `vsink` statements in the disconnected vADP. For each dynamical system variable, this stage synthesizes an assembly fragment that accepts, as input, a signal implementing this variable and produces, as output, a collection of signals which fulfill a subset of `vsink` statements.

3. **Circuit Completion**: This stage completes the disconnected VADP by integrating the generated assembly fragments into the circuit. The resulting completed VADP has no `vsink` statements left, as the integrated assembly fragments have fulfilled them.

8.3.1 Circuit Collation

For each dynamical system variable, the collation procedure accepts as input a set of VADP fragments which implement the dynamics of that variable. Each synthesized fragment contains a single `vsource` statement implementing the dynamical system variable and any number of `vsink` statements which indicate where certain signals are needed in the fragment. Section 8.2 describes the synthesis compilation pass in detail.

The collation procedure nondeterministically selects a VADP fragment for each dynamical system variable and collates these fragments together to form a disconnected circuit. Each dynamical system variable is uniquely implemented by one `vsource` statement in the resulting disconnected circuit. The collation procedure re-assigns block identifiers when merging these fragments to prevent identifier collisions. It produces as output a disconnected VADP made up of fragments that implement the dynamical system.

8.3.2 Assembly Fragment Synthesis Overview

For each dynamical system variable dv, the assembly fragment synthesis procedure (AFSP) produces a VADP assembly fragment $VZ_{asm,dv}$. It accepts as input a signal implementing dv – the `vsource` statement from the disconnected circuit fulfills this signal. The assembly fragment produces a collection of signals that fulfill `vsink` statements in the disconnected circuit. The body of the fragment is made up of assembly blocks $xb \in AssembleBlocks$ that transform and copy the input signal to implement the desired output signals and statements. Note the compiler must copy analog currents (signals of type `current`) to use them multiple times.

The following subsection describes the assembly fragment synthesis procedure for a dynamical system variable dv. The fragment synthesis procedure operates in two stages:

1. **Interface Elicitation**: The AFSP first infers the interface (input and output signals) for the assembly fragment from the disconnected VADP. All produced assembly fragments must implement this interface. The type of the signal tells the compiler what kind of signal should be used to implement the expression.

2. **Fragment Generation**: The AFSP then generates one or more assembly fragments that implement the inferred input-output signal interface. Each fragment contains a `vsink` statement

that accepts the input signal identified in the interface elicitation step and a set of `vsource` statements that implement the output signals identified in the interface elicitation step.

8.3.3 AFSP Interface Elicitation

The AFSP interface elicitation algorithm accepts as input a dynamical system variable dv to assemble and derives the input-output interface (sources and sinks) for the assembly fragment must implement to bridge the ports in the disconnected circuit. The interface elicitation algorithm analyzes the disconnected circuit $\text{vADP}_{disconn}$ to identify the input-output interface of the fragment. The input interface is a single signal s_{in} and the output interface is a multiset of signals $Ms, ...$

Recall that in the disconnected circuit, each dynamical system variable is implemented at exactly one output port. The vADP identifies the output port which implements some dynamical system variable dv with a `vsource` statement. The AFSP must only produce fragments which can accept this signal as input – this is the input interface (s_{src}) of the assembly fragment:

$$s_{src} = \texttt{element}(\texttt{multi}([[\langle \texttt{sigtype}(op), dv\rangle \mid \texttt{vsource}(op, i, dv) \in \text{vADP}_{disconn}]]))$$

It first identifies the $\texttt{vsource}(op, i, dv)$ statement in the disconnected vadp ($\text{vADP}_{disconn}$) which implements the target dynamical system variable dv. The signal at the identified output port op will be provided as input to the assembly fragment. The produced assembly fragment must therefore accept a signal of type $\texttt{sigtype}(op)$ implementing dv – this is the only input provided to the assembly fragment. The source signal elicitation operation always succeeds because the disconnected vADP contains exactly one vADP fragment which implements each dv.

The AFSP next collects all the `vsink` statements in the disconnected vADP which are functions of the dynamical system variable dv. All of these statements must be fulfilled by the produced assembly fragment as they cannot be fulfilled by manipulating a signal which implements a different dynamical system variable. The AFSP derives a multiset of signals which describe the output interface of the produced fragment:

$$M_{sinks} = \texttt{dedup}(\texttt{multi}([[\langle \texttt{sigtype}(ip), de\rangle \mid \texttt{vsink}(ip, i, de) \in VZ_{disconn} \wedge \texttt{vars}(de) = \{dv\}]]))$$

It identifies each sink statement $\texttt{vsink}(ip, i, de)$ in the disconnected vadp ($\text{vADP}_{disconn}$) which requires a signal that is a function of dv. For each of these statements, the produced assembly fragment must generate a signal of the same type $\texttt{sigtype}(ip)$ that implements the required expression de. Some analog signals (e.g. `voltage` and `digital`) signals can be routed to multiple places. The

`dedup` consolidates `voltage` and `digital` signals which can be used multiple times into a multiset element with cardinality one.

Signal Consolidation (`dedup`): The `dedup` function de-duplicates multiple occurances of `voltage` and `digital` signals in the multiset:

$$\forall \langle st, de \rangle \in Signals, \texttt{dedup}(M)\langle st, de \rangle = M\langle st, de \rangle \texttt{ if } st = \texttt{current else } 1$$

The function reduces the count for all digital and voltage signals to one and leaves analog `current` signals intact. The algorithm must preserve the multiplicity of analog currents because they cannot be used more than once and must therefore be copied to be used in multiple places.

8.3.4 AFSP Fragment Generation

The fragment generation procedure synthesizes the assembly fragment that implements the input-output interface inferred in the previous step. It accepts as input the derived interface ($s_{src} = \langle st_{src}, dv \rangle$ and M_{sinks}) and a set of assembly blocks $XB \subseteq AssembleBlocks$. It produces as output a vADP fragment, made up of assembly blocks, that implements the desired interface.

The AFSP internally works with a library of automatically derived *concrete* assembly blocks (CXB_{lib}). I formalize the space of concrete assembly blocks below:

$$cxb \in ConcAsmBlocks = AssembleBlocks \times Modes \times Signals \times \mathbb{P}(Signals)$$

Each concrete assembly block cxb has a fixed mode and accepts a single input signal implementing dv. Assembly blocks may only have one input and may not have any data fields (see Section 5.5). Each concrete block produces a multiset of signals at its output ports. Because concrete assembly blocks are fully specialized, the fragment generation procedure is much simpler than the synthesis phase described in Section 8.2. The assembly procedure, therefore, does not make use of algebraic unification or tableau-based synthesis algorithms introduced in Section 8.2 to construct assembly fragments.

The AFSP first organizes concrete blocks into sequentially organized levels. These levels loosely capture the general structure of the assembly fragment. Each level i is described as a multiset of concrete blocks ($M_{lv,i}$) and an input-output interface implemented as multisets of input and output signals ($M_{in,i}$ and $M_{out,i}$). These level inputs and outputs directly map to the signals at the input and output ports of the concrete blocks in the level. A sequence of levels is *well-formed* if the following properties hold:

Implements Input Interface: The input interface of the topmost level ($M_{in,0}$) contains exactly

one signal $\texttt{element}(M_{in,0}) = \{s_{src}\}$. This property ensures the fragment implements the input interface elicited in the previous step.

Internally Consistent: The AFSP ensures each level's outputs contain the signals required by the next level's input interface. Given an interior level i, the signals the input interface of the interior level $M_{in,i}$ must be satisfied with the signals at the output interfaces of the parent levels $M_{out,0}..M_{out,i-1}$. The output interface signals used to satisfy the input interface of an interior level i are referred to as bound signals. Each pair of levels i and j is associated with a multiset of bound signals $M_{bnd,i,j}$ which contains all the signals routed from level i to level j. The structure is consistent if the following holds:

$$\texttt{dedup}(M_{in,i}) \subseteq \texttt{dedup}(\bigcup_{j=0...i-1} M_{bnd,i,j})$$

The input interface of a fragment level i is considered satisfied if the multiset of signals from preceding levels to level i includes the signals at the input interface. The input interface of level i only needs one copy of a signal if it is a `voltage` or a `digital` signal since the compiler can reuse these signals multiple times. The `dedup` invocation reduces the count of any `voltage` and `digital` signals to one. Any signals within a level j which are not provided to a child level are considered free signals ($M_{free,j}$).

Implements Output Interface: The unused signals in the assembly fragment must implement the output interface elicited in the interface elicitation step:

$$M_{sinks} \subseteq \bigcup_{i \in 0...n} M_{free,i}$$

The multiset of all free variables in the assembly fragment must include the collection of output signals required at the output interface of the fragment.

The AFSP translates the assembly fragment structure into a vADP fragment, inserting vADP connections, sources, and sinks when necessary. The AFSP is always able to generate a vADP fragment implementing the provided interface from this structure. Because the input tree structure ensures that each input signal at each tree level i is satisfied by an output signal on a parent level, the vADP translation procedure does not need to introduce any unexpected vADP signal sinks. Because the tree structure is guaranteed to implement all the required signals as free signals, the produced vADP generates the vADP signal sources necessary to bridge fragments in the disconnected circuit.

Algorithm 4 Algorithm for building corpus of concrete assembly blocks

```
1: # xb: Assembly block to concretize.
2: # s_src: Input signal provided to assembly block.
3: # returns a collection of concrete assembly blocks which accept s_src as input
4: function BuildConcBlock(xb,s_src)
5:     ⟨st_src, dv_src⟩ = s_src
6:     for {m ∈ Modes | m ∈ modes(xb)} do
7:         M = 0
8:         ip = singleton({ip ∈ InputPorts | block(ip) = xb})
9:         st_in = sigtype(ip)
10:        for {op ∈ OutputPorts | block(op) = xb} do
11:            de = portexpr(op, m)[ip/dv_src]
12:            st = sigtype(op)
13:            M = M ∪ ⟨st, de⟩[1]
14:        generate ⟨xb, m, ⟨st_in, dv_src⟩, M⟩
```

Concrete Assembly Block Generation

The AFSP pre-computes the library of concrete assembly blocks cxb_{lib} from the set of assembly blocks XB. Algorithm 4 presents the algorithm which translates an assembly block into a set of concrete assembly blocks. The algorithm accepts as input the assembly block to process (xb) and the signal (s_{src}) expected at the input interface of the fragment. This signal implements the dynamical system variable dv_{src} with a signal of type st_{src}.

The algorithm then generates a concrete assembly block for each mode in the block (lines 6-14). Each concrete assembly block is defined as a tuple containing the corresponding ADS assembly block (xb), the selected mode, the signal accepted at the input port, and the multiset of signals implemented the block output ports when dv_{src} is supplied as input (line 14). The algorithm identifies the type of the signal (st_{in}) accepted at the input port in lines 8-9 and selects the block mode in line 6.

The algorithm derives the multiset of output signals (M) implemented at the assembly block's output ports in lines 10-13. The algorithm derives the concretized dynamics and type of the signal at each output port (lines 11-12). The signal dynamics and signal type fully identify the signal implemented at the output port. The algorithm derives the concretized dynamics by first applying the selected mode and then substituting all occurrences of the input port with the input dynamical system variable (dv_{src}).

Tree Structure Generation

The tree structure generation algorithm accepts as input a library of concrete assembly blocks CXB_{lib} and the elicited input-output interface of the fragment (s_{src} and M_{sinks}). It produces, as output, a sequence of levels that capture the levels in the tree. This tree structure is nondeterministically generated. The topmost level accepts as input the input interface signal s_{src}. For all other

levels, the level's inputs are satisfied by a subset of the preceding levels' outputs. The tree structure produces at its output interface a multiset of freely available signals that satisfy the multiset of provided required signals M_{sinks}.

Feasability: The AFSP determines if it is possible to implement a collection of signals (M) with the available concrete block library:

$$\texttt{feas}(CXB_{lib}, M) = Supp\ M \subseteq Supp \bigcup_{M' \in Sigs} M' \ \texttt{where}\ Sigs = \{M' \mid \langle xb, m, s', M' \rangle \in CXB_{lib}\}$$

For a collection of signals to be feasible, the AFSP must be able to find at least one concrete block which implements each signal in the block library. Therefore the support of the signal multiset M must be a subset of the support of the multiset describing all of the concrete block library output signals.

General Approach The tree generation algorithm recursively builds up the tree structure from the bottom-most level. The compiler initially invokes the algorithm with the multiset of signals that the output interface of the fragment must implement (M_{sinks}). These signals eventually become the free signals in the produced fragment. This procedure uses the BuildLevel routine to generate a candidate level that implements the provided output signals. The BuildLevel routine accepts as input a library of concrete assembly blocks and a sequence of signals to implement and returns a candidate level that implements the provided signals.

The algorithm then restructures this single candidate level into multiple levels with the RestructureLevel routine to minimize the number of input signals required at the topmost level. This routine takes a target level and its required inputs as arguments. It returns a sequence of multiple levels and the multiset of required inputs for the topmost level. The returned multi-level sequence is the restructured, multi-level structure.

Note that this section describes a simplified version of the approach utilized by the compiler. The implemented approach produces multiple assembly fragments and uses versions of the BuildLevel and RestructureLevel routines which return multiple candidate levels and structures.

Algorithm 5 Assembly tree structure generation algorithm

1: # **CXB**: Concrete assembly block library.
2: # s_{src}: Input signal provided to assembly fragment structure.
3: # M_{req}: Multiset of required output signals.
4: # **returns** assembly fragment structure
5: **procedure** BUILDTREESTRUCTURE(CXB_{lib},s_{src},M_{req})
6: **if** $\neg$`feas`(CXB_{lib}, M_{req}) **then return**
7: M_{lv} = BuildLevel(CXB_{lib},M_{req})
8: $\langle M_{in,0}, [[M_{lv,0}, ..., M_{lv,k}]]\rangle$ = RestructureLevel(M_{req},M_{lv})
9: **if** $M_{in,0} = s^1_{src}$ **then**
10: **generate** $[[M_{lv,0}, ..., M_{lv,k}]]$
11: **else**
12: $parentLevels$ = BuildTreeStructure(CXB_{lib},s_{src},$M_{in,0}$)
13: **generate** $parentLevels + [[M_{lv,0}, ..., M_{lv,k}]]$

BUILDTREESTRUCTURE Routine: Algorithm 5 presents the tree structure generation algorithm. The algorithm first derives a level that implements all the required signals (M_{req}) and restructures the produced level into multiple levels ($M_{lv,0}...M_{lv,k}$) to minimize the number of required input signals (lines 3-4). It then checks to see if the input signals of the first restructured level ($M_{lv,0}$) matches the input interface for the fragment. If the topmost level implements the input interface, then the tree structure is complete, and the algorithm returns the produced levels (lines 9-19).

If the topmost level does not implement the provided interface, then the the algorithm recursively generates a parent tree structure that implements the signals required by the topmost level of the derived structure. This recursive call produces a parent tree structure ($parentLevels$) which includes the multiset of freely available signals $M_{in,0}$. The structure generation algorithm uses these signals to satisfy the input interface of the first restructured level $M_{lv,0}$ of the multi-level structure. The algorithm joins the parent structure with the derived multi-level structure to produce the full tree structure (line 13).

Algorithm 6 Assembly tree level generation algorithm

1: # **CXB**: Concrete assembly block library.
2: # M_{req}: Multiset of required output signals.
3: # **returns** assembly fragment structure
4: **function** BUILDLEVEL(CXB_{lib},M_{req})
5: **if** $M_{req} = 0$ **then return** []
6: $cxb = \langle xb, m, s, M\rangle$ = getBest(CXB_{lib},M_{req})
7: $M'_{req} = M_{req} - M$
8: M_{lv} =BuildLevel(CXB_{lib},M'_{req})
9: **generate** $cxb^1 \cup M_{lv}$

BUILDLEVEL Routine: Algorithm 6 presents the BUILDLEVEL routine. The BUILDLEVEL routine accepts a concrete block library CXB_{lib} and a multiset of free signals to implement M_{req} as output signals. The routine returns a single level (sequence of concrete assembly blocks) as output.

The algorithm first selects the best concrete assembly block *cxb* in the block library using the `getBest` function. The `getBest` function returns the concrete block which implements the most output signals. It then computes an updated collection of required free signals M'_{reqs} which exclude the outputs implemented by the selected concrete block. It then recursively invokes BuildLevel on remaining outputs to populate the rest of the level.

Algorithm 7 Level restructuring algorithm

```
1: # CXB: Concrete assembly block library.
2: # M_reqs: Free output signals to preserve. A multiset of signals.
3: # M_lv: Level to restructure. A multiset of concrete assembly blocks.
4: # returns the multiset of input signals required by the restructured levels and the restructured
   levels (as a sequence of concrete assembly block multisets)
5: function RestructureLevel(M_reqs,M_lv)
6:     M_ins, M_outs = 0, 0
7:     for ⟨xb, m, s, M⟩^n ∈ M_lv do
8:         M_ins = M_ins ∪ s^n
9:         for k ∈ 0...n do
10:            M_outs = M_outs ∪ M
11:    [[M_lv,0, ..., M_lv,k]]= Restruct(M_ins,M_outs,M_reqs,M_lv)
12:    M'_ins = 0
13:    for ⟨xb, m, s, M⟩^n in M_lv,0
14:        M'_ins = M'_ins ∪ s^n
15:
16:    return M'_ins, [M_lv,0, ..., M_lv,k]
```

RestructureLevel Routine: Algorithm 7 presents the high-level algorithm for restructuring a tree level. The level restructuring routine accepts as input a single level of concrete assembly blocks and the multiset of required free output signals which the restructured levels must produce. The routine then restructures the concrete assembly blocks into multiple levels to reduce the number of signals required at the topmost level.

The algorithm first derives the input and output signals that describe the input-output interface for the provided level (lines 6-10). It then restructures the level using the Restruct helper function. This function accepts the level inputs and outputs, the level itself, and the required multiset of output signals. It produces a multi-level structure which the required output signals as free signals. The algorithm then derives the multiset of inputs required at the topmost level (M_{ins}) from the structure. It returns both the structure and the new multiset of required input signals.

Algorithm 8 Level restructuring helper function

1: # M_{ins}: Multiset of level input signals.
2: # M_{ins}: Multiset of level output signals.
3: # M_{reqs}: Multiset of free output signals to implement in the restructured levels.
4: # M_{lv}: Level to restructure. A multiset of concrete assembly blocks.
5: # **returns** the rearranged multi-level assembly structure as a sequence of assembly structure levels (concrete assembly block multisets).
6: **procedure** RESTRUCT(M_{ins},M_{outs},M_{reqs},M_{lv})
7: **if** $\mid level \mid = 0$ **then return** [[[]]]
8: $\langle M_{sel}, cxb_{root}\rangle$= bestRoot($level$,$M_{reqs}$,$M_{ins}$)
9: $M'_{lv} = M_{lv} - cxb^1_{root}$
10: $M'_{ins} = M_{ins} - M_{sel}$
11: $M'_{outs} = M_{outs} - M_{sel}$
12: **assert** $M_{reqs} \subseteq M'_{outs}$
13: $[M_{lv,0}, M_{lv,1}, ...]$= Restruct($M'_{ins}$,$M'_{outs}$,$M_{reqs}$, M'_{lv})
14: $M'_{lv,0} = root^1$
15: $M'_{lv,1} = 0$
16: **for** cxb^n_{chl} **in** $M_{lv,0}$
17: **for** $k \in 0...n$ **do**
18: $\langle xb, m, s, M\rangle = cxb_{chl}$
19: **if** $s \in Supp\ M_{sel}$ **then**
20: $M'_{lv,1} = M'_{lv,1} \cup cxb^1_{chl}$
21: $M_{sel} = M_{sel} - s^1$
22: **else**
23: $M'_{lv,0} = M'_{lv,0} \cup cxb^1_{chl}$
24:
25: **return** $[[M'_{lv,0}, M'_{lv,1}, M_{lv,1}, ...]]$

RESTRUCT Routine: Algorithm 8 presents the algorithm implemented by the RESTRUCT helper function. This helper function is responsible for deriving a multi-level structure from a flat level. It accepts as input a multiset of level input signals M_{ins}, a multiset of level output signals M_{outs}, the level to restructure, and a multiset of free output signals M_{req} required at the level output interface.

The algorithm first selects a root block from the set of assembly blocks in the level. The algorithm uses the output signals produced by the selected block to satisfy the input signals required by the other concrete assembly blocks contained in this level. The algorithm uses the `bestRoot` function to select the block to move up to a higher level (line 8). The `bestRoot` function identifies the concrete assembly block whose outputs cover the most level inputs (M_{ins}).

The `bestRoot` function returns both the concrete assembly block (cxb_{root}) and the subset of the selected block's output signals to use when restructuring the level (M_{sel}). The algorithm uses the selected signals to satisfy the input interfaces of a subset of concrete assembly blocks within the new multi-level structure. Because the selected output signals will be internal to the multi-level structure, they will no longer be accessible at either the input or output interface. The algorithm removes the selected signals from the multiset of input signals required at the level's input interface and from the multiset of free signals at the level's output interface (lines 10 and 11). The algorithm

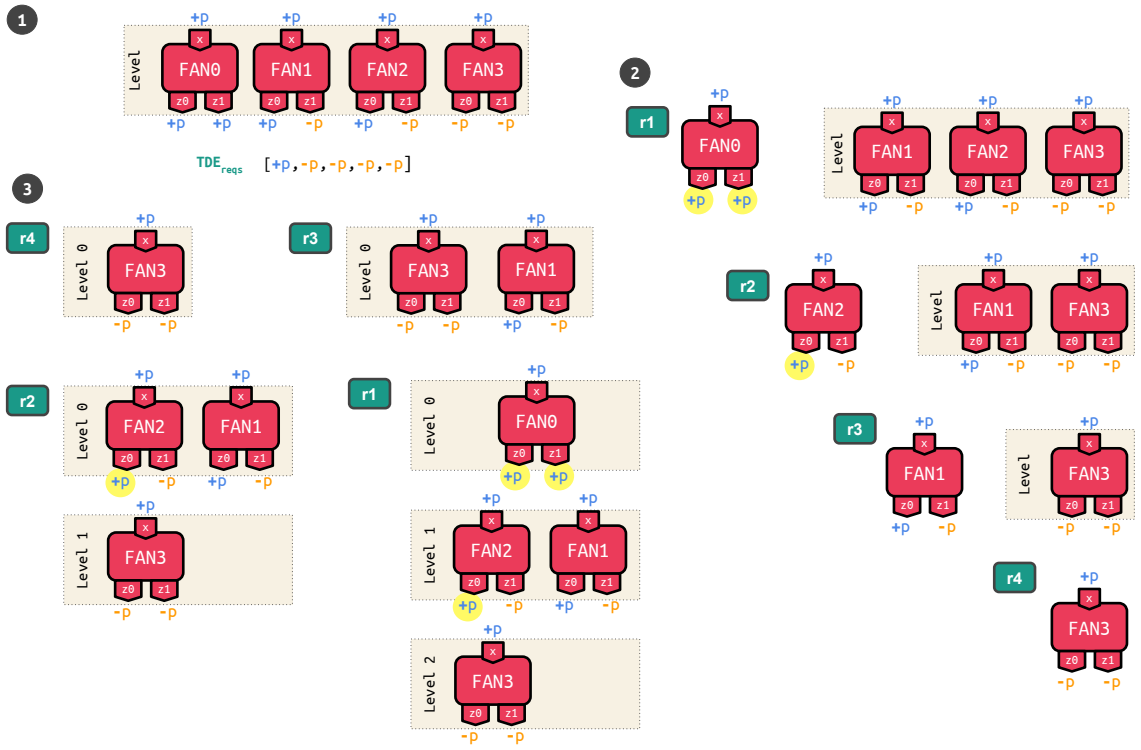


Figure 8-3: Example invocation of tree restructuring algorithm. All signals are of the same type and the modes are omitted.

also updates the provided level to exclude the selected root block (line 9).

The algorithm then recursively calls the RESTRUCT routine with these new multisets of external inputs and outputs to derive the substructure which implements the remaining nodes (line 13). It then creates a new structure where the selected root block is at the topmost (1st) level ($M'_{lv,0}$) and that block's children are at the next (2nd) level ($M'_{lv,1}$). It sorts all the blocks at the topmost level of the derived substructure ($M_{lv,0}$) into either the topmost level or the second level (lines 16-23). For each target block, the algorithm tests if the block's input can be satisfied with one of the selected outputs from the root node. If the target block's input matches one of the selected outputs, the compiler adds the block to the second level. If the algorithm cannot satisfy the input interface of the target block with one of the selected signals, it adds the block to the topmost level.

Illustrative Example: Figure 8-3 presents an example execution of the restructuring algorithm. This algorithm starts with a single level of concrete assembly blocks that implement the required signals. It produces a multi-level structure of rearranged concrete assembly blocks that implement the required signals while requiring fewer outputs.

1. This execution starts with a flat level of four concrete `FAN` blocks which together implement four negative copies and three positive copies of p. The compiler requires all negative copies of p and one positive copy of p to be implemented as free signals in the restructured struc-

ture. This constraint ensures the produced structure produces the signals required to bridge fragments in the disconnected circuit.

2. The algorithm recursively decomposes the input level by repeatedly selecting root blocks and output signals from the level. The algorithm first chooses the `FAN0` block (r1). This block is chosen first because it produces two positive copies of p – the compiler can use these copies to satisfy two signals at the level input interface without dipping into the required signals. The algorithm next chooses the `FAN2` block (r2). The `FAN2` block produces one positive copy of p that can be used to resolve one level input. At this point, the algorithm cannot select any more block outputs without drawing from the multiset of required signals. The algorithm chooses `FAN1` first, and `FAN3` second (r3 and r4). The algorithm does not select any signals for either block since doing so would cause the new structure to fail to implement the required signals at the output interface.

3. The algorithm then builds the multi-level structure by successively adding each selected block. It first returns a single-level structure comprised of a single `FAN3` block (r4). The algorithm then adds the `FAN1` block to the structure (r3). Because the `FAN1` block has no selected signals, the algorithm adds the `FAN3` block to the same level as the `FAN1` block. Next, the algorithm adds the `FAN2` block to the structure (r2). Because the `FAN2` block has one selected positive copy of p, either the `FAN3` and `FAN1` block can be placed underneath `FAN2`. The resulting structure has two levels, an upper level with `FAN2` and `FAN1` blocks and a lower level with a `FAN3` block. Finally, the algorithm adds the `FAN0` block to the structure (r1). The algorithm has previously selected two positive copies of p for this block. The algorithm uses these positive copies of p as input signals for both the `FAN2` and `FAN1` blocks.

The resulting structure has three levels, accepts one positive copy of p, and produces one positive copy of p and four negative copies of p as free signals. The derived structure internally uses three positive copies of p.

Translation to vADP (TreeToVADP)

Algorithm 9 presents the vADP translation algorithm. This algorithm translates the derive assembly fragment structure to a vADP fragment. The vADP translation routine accepts as input a tree structure (`levels`) and returns a vADP fragment which implements the tree structure.

This algorithm works with sets of output port signals $os \in OutputPortSignals = OutputPorts \times \mathbb{N} \times Signals$. The algorithm iterates over the sequence of levels starting with the topmost level. At each level, it generates `vconfig` statements that configure the assembly blocks. If the target level is the topmost level, then the algorithm produces a `vsink` statement for each assembly block input. This `vsink` statement defines the input interface of the fragment. The assembly fragment generation

Algorithm 9 vADP translation algorithm

```
1: # levels: Assembly fragment structure. A sequence of concrete assembly block multisets.
2: # returns a vADP fragment which implements the provided assembly fragment structure.
3: function LevelsToVADP(levels)
4:     let OS_free = ∅
5:     for M_lv,i in levels do
6:         let OS_outs,i = ∅
7:         for ⟨xb, m, s, M⟩^n in M_lv,i do
8:             for k ∈ 0...n do
9:                 let ⟨st, de⟩ = s
10:                let i_new = fresh-id(xb)
11:                let ip = singleton({ip ∈ InputPorts | block(ip) = xb})
12:                let OS = {⟨op, i_new, ⟨sigtype(op), portexpr(m, op)[ip/de]⟩⟩ | block(op) = xb}
13:                let OS_outs,i = OS_outs,i ∪ OS
14:                generate vconfig(xb, i_new, {m}, ∅)
15:                if i = 0 then
16:                    assert | M_lv,i |== 1
17:                    generate vsink(ip, i_new, de)
18:                else
19:                    let ⟨op, i, s'⟩ = SelectFreeOutputPort(OS_free,s)
20:                    let OS_free = OS_free/⟨op, i, s'⟩
21:                    generate vconn(op, i, ip, i_new)
22:        let OS_free = OS_free ∪ OS_outs,i
23:    for ⟨op, i, ⟨st, de⟩⟩ ∈ OS_free do
24:        generate vsource(op, i, de)
```

procedure guarantees the input interface of the produced fragment has exactly one `vsink` statement that accepts the input signal s_{src}.

If the target level is an interior level, then the algorithm inserts `vconn` statements that route free signals from the preceding levels to where they are needed in the current level. The algorithm maintains a set of output ports with freely available outputs (OS_{free}). For each assembly block xb requiring signal s, the algorithm selects an output port that implements s from the set of freely available output ports. The SelectFreeOutputPort function invocation nondeterministically chooses a free output port from OS_{free} which implements the desired signal and returns the chosen output port. The chosen output port becomes the starting port in the `vconn` statement. After each level is processed, the algorithm adds all the output ports from the level to the set of freely available outputs. The algorithm must perform this operation after all of the concrete assembly blocks in the level have been processed to avoid introducing cycles into the vADP fragment.

After processing all the levels, the algorithm translates any remaining free output ports to `vsource` statements. The assembly fragment generation procedure guarantees that these output ports implement the signals required at the output interface of the fragment.

Selecting a Free Output Port: Algorithm 10 presents the algorithm for selecting a free output port. The algorithm accepts as input a set of freely available output port signals and the target

Algorithm 10 vADP output signal selection algorithm

```
1: # OS_free: The set of unused (free) output port signals
2: # s: The target output signal.
3: # returns the chosen output port signal and the set of remaining free signals.
4: function SelectFreeOutputPort(OS_free,s)
5:     let ⟨st, de⟩ = s
6:     let {os} = choose({⟨op, i, s'⟩ ∈ OS_free | s' = s}, 1)
7:     if st = digital ∨ st = voltage then return os, OS_free
8:     else
9:         return os, OS_free/{sig}
```

signal. The algorithm returns the chosen output port signal and the set of remaining free output port signals.

The algorithm nondeterministically chooses an output port signal from the set of free output signals which implement the signal s (line 6). If the output port implements an analog `current`, the algorithm removes the output port from the set of free output ports. This step is necessary because analog currents may only be used once. If the output port implements an analog `voltage` or a `digital` signal, then it is kept in the set of free output ports. The algorithm can use analog voltages and digital signals more than once since routing such signals to multiple places does not compromise the fidelity of the signals.

8.3.5 Assembly Fragment Integration

The compiler generates an assembly fragment $VZ_{asm,dv}$ for each dynamical system variable dv in the target dynamical system. The compiler integrates each assembly fragment into the disconnected circuit ($VZ_{disconn}$). The assembly fragment integration procedure links disparate subcircuits together and resolves `vsink` statements in the disconnected circuit. After the integration procedure, the vADP is fully connected and has no `vsink` statements left.

Algorithm 11 vADP assembly fragment integration algorithm

```
# VZ_asm: vADP assembly fragment
# VZ_disconn: disconnected vADP
# returns a vADP assembly fragment which integrates the provided assembly fragment.
function IntegrateFragment(VZ_asm, VZ_disconn)
    let vsink(ip, i, dv) = singleton({vsink(ip, i, de) ∈ VZ_asm | de = dv})
    let vsource(op, i', dv') = singleton({vsource(op, i, dv) ∈ VZ_disconn | de = dv})
    for vz ∈ VZ_asm/{vsink(ip, i', dv)} do
        generate vz
    generate vconn(op, i, ip, i)
    let sinks = {vsink(ip, i, de) ∈ VZ_disconn | vars(de) = {dv}}
    let srcs = {vsource(op, i, de) ∈ VZ_asm}
    for vz ∈ VZ_disconn/sinks do
        generate vz
    for vsink(ip, i, de) in sinks do
        let vsource(op, i, de'), srcs = selectFreeVADPSource(srcs, sigtype(ip), de)
        generate vconn(op, i', ip, i)
```

Algorithm 11 presents the algorithm for integrating an assembly fragment into the disconnected circuit. The integration algorithm accepts as input the disconnected circuit and the assembly fragment to integrate. It returns the vADP with the integrated assembly fragment as output.

The algorithm first identifies the relevant `vsource` and `vsink` statements in the disconnected vADP. This step is identical to the procedure used to identify the source and sink signals in the interface elicitation step (Section 8.3.3). It first identifies the `vsource` statement in the disconnected vADP which implements the target dynamical system variable dv (`vsource`(op, i, dv)). The AFSP guarantees that the assembly fragment contains a single `vsink` statement which accepts a signal of type `sigtype`(op) implementing dv – the exact signal produced by the identified `vsource` statement.

The algorithm inserts a `vconn` which connects the identified source and sink statements. At this point, the identified `vsink` statement is satisfied and can be excluded from the generated vADP. Next, the algorithm identifies all the `vsink` statements in the disconnected circuit that must be satisfied by the target assembly fragment. It then identifies a `vsource` statement in the assembly fragment for each `vsink` statement in the disconnected circuit and produces a `vconn` statement bridging the identified source and sink. The *selectVADPSource* function nondeterministically chooses a `vsource` statement which implements the signal required by the `vsink` statement and updates the

list of free vADP sources (*srcs*) if necessary. The vADP returned by the integration procedure contains no sink statements involving the dynamical system variable dv.

Algorithm 12 VADP source signal selection algorithm

```
1: # free: set of free vsource statements
2: # st: type of target signal.
3: # de: dynamical system expression implemented by target signal.
4: # returns the chosen vsource statement and the updated set of unused (free) vsource statements
5: procedure SelectFreeVADPSource(free,st,de)
6:     let {sig} = choose({vsource(op, i, de') ∈ free | de' = de ∧ sigtype(op) = st}, 1)
7:     if st = digital ∨ st = voltage then return sig, free
8:     elsereturn sig, free/{sig}
```

Selecting a Free `vsource`: Algorithm 12 presents the vADP signal selection algorithm. Given a set of unused (free) `vsource`statements and the desired signal, the algorithm nondeterministically selects a free `vsource` statement which implements the signal:

The algorithm nondeterministically chooses an output port that implements the desired dynamical system expression using a signal of the desired type is from the set of free vADP sources (line 6). If the output port implements an analog `current`, then it is removed from the set of free vADP sources. This step is necessary because analog currents may only be used once. If the output port implements an analog `voltage` or a `digital` signal, then it is kept in the set of free vADP sources. Because the compiler can use such signals more than once, the signals are not removed from the set of free sources after the algorithm chooses them.

8.4 Place and Route

The place and route (P+R) stage of compilation maps vADP block instances $\langle b, i\rangle$ to block locations $l \in Locs$, inserting route blocks $rb \in RouteBlocks$ as necessary. The place and route procedure accepts as input the completed vADP and the device layout specification from the ADS (see Section 5.5). The place and route procedure resolves all virtual block instances to locations on the hardware and maps connections to sequences of digitally programmable interconnects available in hardware. It produces an analog device program that implements the completed vADP on the target hardware.

8.4.1 Placement

`LGraph` incorporates a *spatially aware* placement algorithm that places interconnected blocks close to each other in the device. It incrementally resolves the location of each block in the circuit by assigning it to spatial locations in increasingly specific views. Recall the hardware specification

organizes the device into sequentially organized spatial views [[1...i...n]]. Each view contains multiple spatial locations that capture spatial regions where blocks are co-located on the analog device. The spatial locations on the most specific view (n) are the unique block locations that identify block instances.

The placement algorithm breaks down the block placement problem into several smaller, more manageable block placement problems (BPPs). The algorithm incrementally refines the spatial location of each block instance by placing it at a location in each view. The algorithm first solves the BPP for the most general (root) view. It then iteratively solves the BPP for each subsequent view, using the location assignments for the parent view as restricting assignments for the BPP. If the algorithm cannot find a set of satisfying assignments for a view, the algorithm backtracks and finds alternate assignments for a parent view. Note that this algorithm doesn't need to backtrack often, as early placement decisions assign blocks to larger structures in the device that are difficult to connect together. The produced location assignments place densely connected sets of blocks within the same spatial region of the chip. This placement strategy is desirable as analog device micro-architectures typically prioritize providing programmable connections between blocks that are spatially co-located.

Note that the placement algorithm takes longer assigning locations to vADPs that implement large dynamical systems where variables are used many times. The placement algorithm must densely pack the blocks at every view in these systems to avoid exhausting the available route blocks. As a result, the algorithm has to backtrack more often, causing performance degradations.

Block Placement Problem (BPP)

The block placement problem (BPP) is an integer linear programming problem that assigns virtual block instances from a vADP to locations in a target spatial view sv, subject to a set of restricting spatial location assignments for the preceding view. The BPP nondeterministically produces a set of satisfying spatial location assignments which are consistent with the restricting assignments and respect the block instance and connectivity limitations imposed by the ADS. A set of assignments is *consistent* if each assigned location is a child of its respective restricting location.

The BPP is made up of a collection of constraints over binary membership variables [110]. The BPP contains three types of binary membership variables:

- **Instance Variables**: Instance variables $\texttt{b-inst}(b, i, sl_{sv})$ assign vADP block instances to locations in the view. The location assignments are derived from the set of enabled (set to 1) instance variables.
- **Path Variables**: Path variables $\texttt{b-path}(\texttt{vconn}(op, i, ip, i'), id)$ assign vADP connections to distinct paths on the device. Each BPP path is associated with a starting block location

(b, sl_{sv}), ending block location (b', sl'_{sv}) (where $b, b' \in AssembleBlocks \cup ComputeBlocks$). Each path may directly connect two blocks or indirectly route the signal through a set of route block locations $\{(rb, sl_{sv}) \in SpatBlockInst_{sv} \mid rb \in RB\}$.

The BPP only models paths between distinct locations because analog devices typically provide fewer connections between spatially distant blocks (blocks assigned to different locations) than spatially co-located blocks (blocks assigned to the same location). This simplification reduces the complexity of the vADP while still producing a set of routable placements.

- **Route Block Variables**: Route block variables `b-rb`(id, rb, sl_{sv}) assign route block locations to path identifiers id. If a BPP path contains one or more route blocks, then the route block variables for that path are enabled. The BPP uses these variables to ensure that the same route block isn't used for multiple distinct paths.

8.4.2 Routing

The routing algorithm maps connections in the vADP to paths in the ADS. Each path contains one or more digitally programmable connections that are linked together with route blocks. For each connection, the routing algorithm assigns a candidate path that implements the connection. After each path assignment, the algorithm removes any intersecting paths from the set of candidate paths. If no viable candidate paths are left, the algorithm backtracks and selects a different path assignment for an earlier connection. This lightweight algorithm is often able to find satisfying routing solutions because the BPPs solved during placement encode coarse-grain routing restrictions into the placement problem.

8.4.3 Block Placement Problem Generation

The BPP solver generates a block placement problem for a spatial view sv from the input vADP, the ADS, and a set of restricting assignments BL_{sv-1}. It produces a set of BPP constraints, that when solved with an ILP solver, produce a set of spatial location assignments that meet the criteria outlined in Section 8.4.1. The generated constraints are described below:

Block-Location Assignment: Each block instance in the vADP is assigned to exactly one location in the spatial view. For example, given a block configuration statement `vconfig`(b, i, M, A), the sum of all BPP instance assignments must equal exactly one:

$$\sum_{sl_{sv}} \texttt{b-inst}(b, i, sl_{sv}) = 1 \tag{8.1}$$

Block Availability: The number of block instances mapped to each location in the ADS does not exceed the available number of blocks at that location. For example the block b at location sl_{sv} in

view sv has $|\{sl_n \mid \texttt{spat-member}(sl_n, sl_{spatview})\}|$ block instances at the most specific view on the chip. Therefore, the sum of all BPP instance assignments involving sl_{sv} must be less than or equal to the total number of available instances:

$$\sum_i \texttt{b-inst}(b, i, sl_{spatview}) \leq |\{sl_n \mid \texttt{spat-member}(sl_n, sl_{spatview})\}| \tag{8.2}$$

Path Assignment: Each $\texttt{vconn}(op, i, ip, i')$ statement must be assigned to exactly one path on the analog device:

$$\sum_{id} \texttt{b-path}(\texttt{vconn}(op, i, ip, i'), id) = 1 \tag{8.3}$$

Note that the BPP only encodes path assignments which bridge the vconn input and output blocks. In the above example, it will only consider paths which connect $\texttt{block}(op)$ blocks to $\texttt{block}(ip)$ blocks.

Block Instance Consistency: Each path assignment must be consistent with the instance assignments made to the source and destination virtual block instances. For example, if a vADP connection $\texttt{vconn}(op, i, ip, i')$ is assigned to a BPP path id which connects $(\texttt{block}(op), sl_{sv})$ to $(\texttt{block}(ip), sl'_{sv})$, then the virtual block instances must be assigned the to the appropriate block locations:

$$\texttt{b-path}(\texttt{vconn}(op, i, ip, i'), id) = \texttt{b-inst}(\texttt{block}(op), i, sl_{sv}) \wedge \texttt{b-inst}(\texttt{block}(ip), i', sl'_{sv}) \tag{8.4}$$

Route Block Utilization: Each path assignment must utilize all the route blocks which appear in the path. For example, if BPP path with identifier id is in use, each route block location (rb, sl_{sv}) which appears on the path must marked as enabled:

$$\sum_{\texttt{vconn}(op,i,ip,i')} \texttt{b-path}(\texttt{vconn}(op, i, ip, i'), id) = \texttt{b-rb}(id, rb, sl_{sv}) \tag{8.5}$$

Route Block Availability: Each route block may only be used once. A route block rb at location sl_{sv} must have at most one route block assignment active at a time:

$$\sum_{id} \texttt{b-rb}(id, rb, sl_{sv}) \leq 1 \tag{8.6}$$

This constraint ensures that no two utilized indirect paths intersect.

Restricting Assignments: If a block has a restricting location assignment, the block can only be assigned to child locations of the restricting location in the target view. If some virtual block instance (b, i) has been previously assigned to location sl_{sv-1} by a preceding block placement operation, it

can only be assigned to spatial locations which are contained by sl_{sv-1}.

$$\sum_{sl_{sv}} \texttt{b-inst}(b, ident, sl_{sv}) = 0 \texttt{ if } \neg\texttt{spat-member}(sl_{sv}, sl_{sv-1}) \tag{8.7}$$

8.4.4 Place and Route Algorithm

The place and route algorithm operates by progressively assigning a location sl_{sv} to each virtual block instance (b, i) for each spatial view $1...sv..n$. Given a spatial view sv, each block-location assignment bl_{sv} consists of a virtual block instance (block and identifier) and a spatial location – a tuple of integers – belonging to spatial view sv:

$$bl_{sv} \in BlockLocAssigns_{sv} = Blocks \times \mathbb{N} \times SpatialLocs_{sv}$$

I use the notation BL_{sv} to refer to a set of block-location assignments for spatial view sv.

Algorithm 13 Place and route algorithm

```
# sv: Current spatial view to assign blocks to.
# VZ: The vADP to map to the hardware.
# BL_{sv-1}: The set of spatial location assignments from the previous spatial view.
# returns a set of ADPs which implement the vADP on the target hardware.
function PlaceAndRoute((sv,VZ,BL_{sv-1}))
    if sv − 1 = n then
        generate Route(BL_{sv},VZ)
    else
        for BL_{sv} ∈ SolveBPP(sv,VZ,BL_{sv-1}) do
            for Z ∈ PlaceAndRoute(sv + 1,VZ,BL_{sv}) do
                generate Z
```

Algorithm 13 presents the place and route algorithm. The algorithm accepts as input a vADP to process, a spatial view sv to target, and a set of block-location assignments BL_{sv-1} made to the preceding view. It produces multiple analog device programs Z that implement the provided vADP on the analog hardware.

If the preceding view $(sv - 1)$ is the most specific view in the analog device, the algorithm translates the vADP to an ADP using the assignments made to the preceding view. Otherwise, the placement algorithm solves a block placement problem to compute the set of block-location assignments for spatial view sv. The *SolveBPP* function solves the block placement problem for the view, given the set of restricting spatial location assignments BL_{sv-1}. It then recursively calls on `PlaceAndRoute` to complete the assignment process for subsequent views for each set of computed block-location assignments BL_{sv}.

8.5 Conclusion

In this chapter, I presented a rigorous description of the circuit synthesis pass. The circuit synthesis pass of compilation synthesizes an analog circuit comprised of configured blocks that implement a target dynamical system. This compilation pass works with an idealized abstraction of the hardware platform and does not consider any low-level physical effects and physical behaviors present in the device. The circuit synthesis pass takes, as input, an analog device specification (ADS) and a dynamical system specification (DSS). It returns an analog device program that implements the provided dynamical system on the analog device described in the ADS. The circuit internally works with a virtual analog device program (vADP) representation that supports defining circuit fragments. The vADP representation identifies blocks with numerical identifiers instead of device locations.

I first introduce all of the necessary notation for accessing information from the ADS and DSS and present the mathematical constructs for defining ADPs and vADPs. I also introduce notation for non-deterministic operators, multisets, mathematical expressions, and variable assignments. I then describe each step of circuit synthesis using this notation.

Fragment Synthesis: The circuit synthesis pass first synthesizes a collection of vADP fragments which each implement a target dynamical system relation. This step of synthesis accepts as input the analog device specification and the dynamical system relation to implement and produces a set of vADP fragments that implement the dynamical system relation as output. Each produced vADP fragment routes together a collection of configured analog compute blocks together to form a partial circuit. The fragment synthesis procedure annotates the vADP fragment's input and output interface with signal sink and source annotations, respectively. The fragment synthesis procedure employs a tableau-based synthesis algorithm to construct fragments. Each tableau contains a set of goals, a set of available hardware relations, and the vADP currently being derived. The algorithm begins with an initial tableau containing and repeatedly derives tableaus from this starting tableau until it identifies a solved tableau. The fragment synthesis procedure returns the vADP from the solved tableau.

The synthesis algorithm uses the tableau transition ($\rightarrow$) operator to derive a new tableau from a seed tableau. The tableau transition operator selects a compatible goal and hardware relation from the tableau and then unifies the goal with the hardware relation to derive a block configuration that solves the goal with the selected hardware relation. The unification algorithm derives a set of port- and data field assignments that render the hardware relation algebraically equivalent to the chosen goal when substituted into the hardware relation. The unification algorithm makes use of a computer algebra system to identify non-trivial unifications. The transition operator then updates the tableau vADP, goals, and hardware relations to reflect the results of the unification. After describing the

basic operation of the fragment synthesis procedure, I describe the extension to fragment synthesis that enables the compiler to use physical laws such as Kirchoff's law to perform computation.

Assembly: The circuit synthesis pass then assembles together the vADP fragments to form a completed circuit. The assembly procedure first selects a vADP fragment that implements each dynamical system relation and collates the fragments together to form a disconnected circuit comprised of circuit fragments. The assembly procedure synthesizes vADP fragments comprised of assembly blocks that interface between disconnected vADP fragments in the disconnected circuit. The assembly procedure employs a specialized form of fragment synthesis which specifically targets copier and conversion blocks, which do not implement complex computations. The assembly procedure first identifies the input-output interface for the assembly fragment by analyzing the disconnected circuit. It then produces assembly fragments that implement the derived input-output interface. The compiler produces assembly fragments in two stages – the compiler first produces a sketch of the assembly fragment, called the assembly fragment structure, that captures the general topology of the fragment. The compiler then translates the fragment structure into a vADP fragment – the vADP fragments are integrated into the disconnected vADP to produce a fully connected vADP that implements the dynamical system.

Place+Route: The circuit synthesis pass then maps vADP block instances to locations on the analog device, inserting route blocks when necessary. The place+route procedure accepts a fully connected vADP as input and produces an ADP as output. The place+route procedure frames the mapping problem as a series of integer linear programming problems. Each integer linear programming problem maps the blocks in the vADP to increasingly fine-grained structures (spatial locations) on the device. The placement procedure leverages the insight that, in reconfigurable analog devices, there are typically fewer connections between coarse-grained structures than fine-grained structures. Coarse-grained structures also contain more blocks of each type than fine-grained structures since fine-grained structures are enclosed in coarse-grained structures.

I introduce the block placement problem, an ILP problem that encodes the block availability and routing constraints present within the hardware. The block placement problem ensures the blocks assigned to each spatial location do not exceed the total number of available blocks at that spatial location. The block placement problem also ensures that there are enough distinct paths between spatial locations to implement the vADP connections. Once the place+route algorithm derives a set of vADP block instance-location assignments, it then translates the vADP to an ADP, inserting route blocks as necessary to indirectly make connections.

Chapter 9

Analog Circuit Scaling

Chapter 8 treated the available analog blocks as idealized computational units that implement abstract algebraic functions. Analog devices, including the HCDCv2, exploit the device physics to directly implement computation with physical signals such as analog currents. While this direct computation is the key to the energy efficiency of analog devices, it also requires computations to operate successfully in the presence of challenging physical phenomena such as noise, quantization error, process variations, and frequency limitations.

The compiler manages these challenges by scaling the computation to respect the physical limitations and behaviors of the device. This transform respects all of the imposed voltage, current, and frequency limitations and compensates for variations within the device and delivers acceptably accurate computations in the presence of noise and quantization error. This scaled computation ensures the original dynamical system dynamics are recoverable at runtime by scaling the time and magnitude of the collected signals by a compiler-derived inverting transform. Refer to Chapter 6 for a primer on how the scaling transform works and Chapter 7.5 for a high-level overview on how the circuit scaling procedure generates the scaling transform.

The compiler computes a scaling transform to scale the target unscaled computation. This scaling transform is made up of magnitude scale factors which scale the amplitude of each data field and signal in the target circuit and a time scale factor that changes the speed of the simulation on hardware. This compiler derives a scaling transform which is guaranteed to have the following properties:

- **Physically Realizable**: Each of the signals in the scaled ADP respects the operating range and frequency limitations of the port accepting the signal.
- **Recoverable**: The original simulation can be recovered by scaling the magnitudes and times recorded at the sampled digital outputs by derived scaling factors.

- **Good Quality**: The scaled circuit minimizes the effect of quantization error and analog noise on the computation and compensates for the effects of any manufacturing variations in the device.

The `LScale` compilation pass is responsible for computing the best scaling transform for an unscaled circuit. Because so many of these physical limitations and behaviors are influenced by the mode of the block, the `LScale` compilation pass also changes the block modes in the ADP when beneficial. `LScale` formulates the problem of identifying a scaling transform and set of mode selections which deliver the above criteria as a constraint satisfaction problem – specifically a combinational geometric program (CGP). When solved, this problem produces a set of mode selections and scaling factors that deliver a physically realizable and recoverable simulation when applied to the digital inputs of the analog device.

Chapter Summary: In this chapter, I more formally describe the operation of the `LScale` compilation pass. This chapter covers the following topics:

- **The CGP and GP (Section 9.1)**: I formally introduce the basic combinational geometric program and geometric program formulations. I demonstrate that a combinational geometric program reduces to a geometric program (GP) when all discrete variables are concretized. The geometric programming problem can then be converted to a convex optimization problem and solved using a convex solver. The resulting solution is optimal with respect to the chosen objective function.

- **CGP Generation Procedure (Section 9.3)**: I formally introduce all of the notation for describing the `LScale` CGP and present the algorithm for deriving the CGP from the input unscaled ADP. This constraint generation procedure ensures that any satisfying scaling transform renders the ADP physically realizable, of good quality, and recoverable.

- **CGP Factor Constraint Generation (Section 9.4)**: I present the algorithm for deriving the constraints which ensure . The produced factor constraints ensure that the signal of each output port is recoverable, provided all the block inputs and data fields are also recoverable. The factor constraint generation procedure includes optimizations that enable the compiler to compensate for behavioral variations and more aggressively change the block modes. The factor constraint generation algorithm works with master expressions that capture the correctable behavioral variations and the model the effect of mode changes on the block behavior. The factor constraints are incorporated into the CGP by the CGP generation procedure.

- **Scaling the ADP (Section 9.5)**: The CGP generation procedure tractably compensates for process variations present in the device on hand by deriving a symbolic expression for each ADP output port which captures the full range of empirically observed behaviors. This

symbolic expression is referred to as the master expression of the output port in this thesis. The CGP generation procedure uses the master expression during CGP generation to produce constraints that ensure the original dynamics of the dynamical system are recoverable at each output port. I formally describe the the algorithm used for eliciting this symbolic expression from the observed empirical behavior of the block.

9.1 The CGP and GP

The CGP is a type of mixed-integer constraint problem which contains both integer-valued ($k \in \mathbb{I}$) and positive, non-zero real-valued variables ($y^+ \in \mathbb{R}^+$). The CGP supports a number of non-linear constraints over its real-valued variables and a highly restrictive set of implication and disjunction constraints over its integer-valued variables. The constraints over the integer-valued variables are used to relate the integer variable values to real variable values or limit the set of values an integer-valued variable can take on. A special property of the CGP is that becomes a geometric programming problem when the all of the integer-valued variables are concretized. A geometric programming problem is a type of constrained optimization problem which can be solved with a convex optimizer [18, 92].

For this reason, the CGP is typically solved twice. The CGP is first directly solved with a SMT solver to obtain a set of viable integer variable and real variable assignments. The real variable assignments are discarded and the integer variable assignments are applied to the CGP to produce a GP. This procedure eliminates all integer variables and constraints, leaving only geometric constraints. These geometric constraints can then be paired with an geometric programming objective function to find an optimal set of geometric variable assignments with respect to some metric.

9.1.1 The Geometric Programming Problem

A geometric programming problem consists of an objective function po_{opt} and constraints $gc \in GeometricConstraints$ over the geometric program variables $GeometricVars$. The geometric program may contain inequality constraints $mo_i \leq 1 \in GeometricConstraints$ and equality constraints $po_j = 1 \in GeometricConstraints$:

$$minimize \;\; s_{opt}$$

$$mo_i \leq 1, i = 1, ..., n$$

$$po_j = 1, j = 1, ..., m$$

Here the mo are *monomials* of the following form, where $y_i^+ \in \mathbb{R}^+$ (positive real numbers), $gv \in$

$GeometricVars$, and $x_i \in \mathbb{R}$:

$$m_i = y_i \prod_{j \in 0..n} gv_{i,j}^{x_{i,j}}$$

Monomials are closed under multiplication and exponentiation to a real number, and are therefore closed under division by extension. Any positive nonzero constant is a monomial. The po_j are *posynomials* (sums of monomials) of the following form:

$$po_i = \sum_i mo_i = \sum_i y_i^+ \prod_{j \in 0..n} gv_{i,j}^{x_{i,j}}$$

The variables in a geometric program (i.e., $gv \in GeometricVars$) take on only positive, non-zero values. The geometric program solver first transforms the geometric programming problem into a linear programming problem by taking the logarithm of the constraints and objective function. It then solves the linear programming problem with a convex optimizer to find the optimal set of assignments.

9.1.2 The Combinatorial Geometric Programming Problem

The combinatorial geometric programming problem (CGP) is a constraint problem which contains both geometric program constraints ($gc \in GC$) over geometric program variables and limited set integer constraints over geometric program variables ($gv \in GeometricVars$) and integer variables ($iv \in IntegerVars$). It supports two kinds of integer constraints:

Geometric Program Variable Assignment Constraints: The CGP introduces variable assignment constraints which set the value of geometric programming variables depending on the value of an integer variable:

$$iv = k \implies gv = y^+$$

Integer Variable Limitation Constraints: The CGP introduces constraints which limit the set of values each integer variable may take on:

$$\bigvee_{0..i..N} iv = k_i$$

Integer Variable Exclusion Constraints: The CGP introduces constraints which disallow integer variables from taking on certain integer values.

$$iv \neq k$$

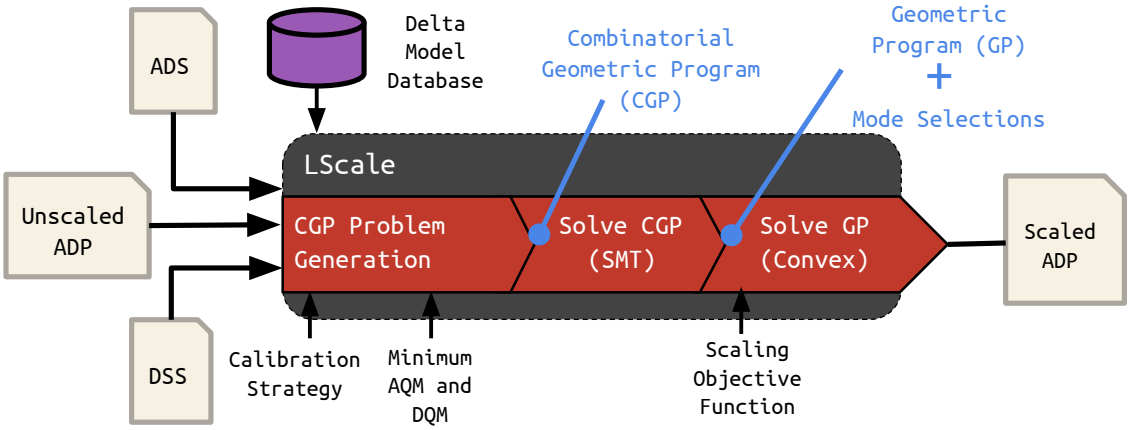


Figure 9-1: Overview of circuit scaling (`LScale`) pass

9.2 Problem Definition

`LScale` accepts a dynamical system specification (DSS), analog device specification (ADS), an analog device program (ADP), the scaling objective function to minimize, an empirically-derived delta model database, and the calibration strategy to target. The `LScale` pass also optionally accepts a a minimum analog and digital quality measure (AQMmin and DQMmin). The delta model database captures all of the device-specific behavioral deviations observed in the calibrated blocks for the device on hand. The calibration strategy specifies how the blocks are calibrated. Refer to Section 5 for a detailed discussion on delta models and device calibration.

`LScale` produces as output a scaled analog device program where each block is assigned exactly one mode. This scaled analog device program contains a collection of magnitude scaling factors (defined with `scf` statements) and a time scaling factor (defined with a `timescale` statement). The computed scaling transform and mode assignments transform the signals so that they abide by the operating range limitations and frequency limitations described in the ADS and compensate for any unexpected correctable signal gains described in the delta model database. All scaled signals meet the quality requirements specified by the user-provided AQMmin and DQMmin. The scaled ADP ensures that the original dynamics of the input DSS can be recovered by scaling the signal by scaling factors and time constant. The resulting scaling transform is also optimal and minimizes the user-provided objective function. The scaling procedure also computes a set of *injection variables* which modify the expressions implemented at expression data fields in the ADP. These injection variables enable `LScale` to more flexibly scale the ADP.

`LScale` produces this scaled analog device program in two stages. First it derives a combinatorial geometric program (CGP) from the ADP, ADS, DSS, minimum quality measures (AQMmin, DQMmin), calibration strategy, and delta model database. This CGP contains both geometric programming constraints and SMT constraints. The geometric programming constraints ensure the

produced scaling transform is physically realizable and recoverable. The SMT constraints encode the effect of changing the block mode on the geometric constraint problem. The CGP is solved with an SMT solver to produce a geometric program and a set of mode assignments. This geometric program constraints are then paired with a scaling objective function to form a convex optimization problem. This convex optimization problem is then solved to find the scaling transform that minimizes the scaling objective function – this is the optimal scaling transform. The scaling transform, mode selections, and injection variables are then incorporated into the input ADP to produce the scaled ADP (Section 9.5).

9.2.1 Notation

This chapter reuses the set notation and notations for mathematical expressions, real numbers, and natural numbers presented in Section 8.1. It also introduces several new mathematical constructs in this chapter.

Many of the techniques presented in this chapter work with positive, nonzero, real values. I use the notation $y^+ \in \mathbb{R}^+$ to express positive, nonzero reals. Intervals $[y, y'] = ival \in Intervals = \mathbb{R} \times \mathbb{R}$ describe ranges of values. The first number is the lower bound of the interval, and the second number is the upper bound. The lower bound is always less than or equal to the upper bound. If the lower and upper bound of an interval is equal, then the interval describes a constant value.

I also introduce a factor function `factor`. This function takes an algebraic expression as input and decomposes the expression into a positive constant coefficient and a base expression. If $\texttt{factor}(e)$ returns $\langle y^+, e' \rangle$ then $e \equiv y^+ \times e'$.

9.2.2 Analog Device Specification

This chapter uses the basic notation presented in Section 8.1 of this thesis. Specifically, it reuses the notation for block modes ($m \in Modes$), input and output ports ($ip \in InputPorts$, $op \in OutputPorts$, and $p \in Ports$), and block instance locations ($l \in Locs$). It uses the `block` : $Ports \to Blocks$ function to retrieve the block which contains a port or data field and the `portexpr` : $OutputPorts \times Modes \to HwExprs$ function to retrieve the expression implemented at each output port. It introduces several new notational constructs which capture the physical limitations and behaviors of the analog device:

- ***Quantization Error***: The quantization error is the maximum possible error for a digital value at a data field or digital port. It corresponds to the maximum distance between two consecutive digital decimal values. This distance between values is automatically computed from the `quantize` statements in the ADS by dividing the range of the digital port by the

number of values. The `quantize-error` : $OutputPorts \rightarrow \mathbb{R}^+$ function maps block output ports and modes to quantization error values. This function is derived from the `quantize` statements in each block specification.

- ***Noise***: Each `noise` property is the standard deviation of the analog noise for an analog signal at an analog port. It captures the signal fluctuations experienced in the analog substrate and is automatically computed from `noise` statements in the ADS. The `noise` : $OutputPorts \rightarrow \mathbb{R}^+$ function maps block output ports and modes to noise values. This function is derived from `noise` statements in each block specification.

- ***Maximum Frequency***: Each `max-freq` property captures the maximum supported speed of the mapped computation. These properties automatically computed from `maxfreq` statements in the ADS. The `max-freq` : $OutputPorts \rightarrow \mathbb{R}^+$ function maps block output ports and modes to maximum frequencies. This function is derived from maxfreq statements in each block specification.

- ***Operating Range***: Each `op-range` property captures the range of analog or digital values supported by a port or data field. These properties are automatically computed from `interval` annotations in the ADS. The `oprangefn` : $(Ports \cup DataFields) \rightarrow Intervals$ function maps block ports,data fields and modes to intervals. This function is derived from maxfreq statements in each block specification.

- ***Time Constant***: I also reference the hardware time constant (`tc`) in this chapter. The `freq` statement in the device layout specification defines the hardware time constant. The time constant describes the mapping between wall clock time and hardware time. One unit of hardware time corresponds to tc^{-1} seconds of wall clock time.

- ***Delta Model Specification***: The delta model specification defines a set of mode-dependent expressions over block ports $p \in ports$, data fields $df \in DataFields$, and delta model parameters $dp \in DP$. The delta model specification for an output port is retrieved with the following function: `delta-spec` : $OutputPorts \times Modes \rightarrow Exprs$. Each delta model specification is uniquely identified by the block output port and block mode.

  Each delta model parameter may be correctable or uncorrectable ($DPType = \{$`correctable`, `uncorrectable`$\}$) and is associated with an ideal value $y \in \mathbb{R}$. The ADS annotates correctable delta model parameters with a hint $DPAnnots = \{$`gain`, `offset`, `other`$\}$ which indicates what kind of behavioral deviation the parameter captures. The delta parameter information function `delta-par-info` : $Blocks \times DP \rightarrow \mathbb{R} \times DPType \times DPAnnots$ returns the information for a delta model parameter.

9.2.3 Dynamical System Specification and Analog Device Program

This chapter uses the interval annotations included in the dynamical system specification and the structure of the analog device program to derive the dynamic range of the signal over each wire. `LScale` derives an interval for each port and data field by mapping the DSS `interval` annotations on the ADP and then performing interval propagation to derive all the other unscaled signal intervals. The `signal-ival` : $Ports \cup DataFields \times Locs \rightarrow Intervals$ function returns the interval of the signal at a particular port or data field in the ADP. Refer to Section A.1 of the Appendix for details on the interval propagation algorithm.

`LScale` also pre-computes several quantities related to real-time execution from the dynamical system specification. `LScale` derives the maximum defined external signal frequency `max-extern-freq` for all of the defined DSS `freq` annotations. `LScale` also records and the the minimum real-time execution speed `min-realtime-speed` and maximum real-time execution speed `max-realtime-speed` from the `realtime` annotation in the DSS.

This chapter reuses the ADP notation presented in Section 8.1. Specifically, it works with the `config`(b, l, M, A) and `conn`(op, l, ip, l') statements in the ADP in its formalization the scaling transform.

9.2.4 Delta Model Database and Calibration Strategy

The `LScale` pass uses delta models to compensate for the behavioral variations in the calibrated blocks. A delta model is an algebraic expression that describes the signal's empirically observed behavior at an output port instance when the block is under a particular mode. Refer to Chapter 5 for more information on delta models. The `LScale` pass accepts as input the calibration strategy $cs \in CalibStrategies$ to target. The `LScale` pass requires the calibration strategy as input because the behavioral variations, and therefore the delta model, change depending on how the block is calibrated. The `LScale` pass constructs the delta models from the delta model parameters in the delta model database and delta model specification from the ADS:

Delta Model Database: The delta model database is a repository of all empirically derived delta model parameters for the calibrated block instances for the device on hand. The delta model parameters for the a particular output port instance is retrieved with the following function: `delta-db` : $CalibStrategies \times OutputPorts \times Locs \times Modes \rightarrow \mathbb{P}(DP \times \mathbb{R})$. This function accepts the target calibration strategy, the output port instance, and the target block mode as input and returns a set of delta model parameter assignments. The returned delta model parameter assignments can be substituted into the delta model specification in the ADS to obtain a delta model which describes the empirically observed behavior of the target output port instance.

Algorithm 14 Delta model retrieval function (`delta-model`)

```
1: function DELTA-MODEL(cs, op, l, m)
2:     e = delta-spec(op, m)
3:     for ⟨dp, y⟩ ∈ delta-db(cs, op, l, m) do
4:         ⟨y′, dptype, dpannot⟩ = delta-par-info(block(op), dp)
5:         if dptype = correctable ∧ dpannot = gain then
6:             e=sub(e, {⟨dp, y⟩})
7:         else
8:             e=sub(e, {⟨dp, y′⟩})
       return e
```

The Delta Model Retrieval Function: The delta model retrieval function `delta-model` : $CalibStrategies \times OutputPorts \times Locs \times Modes \rightarrow Exprs$ returns a delta model expression which captures the behavioral variations that can be compensated for by the `LScale` compilation pass.

Figure 14 presents the delta model retrieval function. The `delta-model` function accepts the output port instance, the block mode, and the calibration strategy to target as input and returns a delta model expression containing only ports and data fields. The delta model expression only allows the delta model parameters that implement correctable gains to deviate from their ideal values. The scaling transform computed in the `LScale` pass can only compensate for correctable gains in the delta model.

The algorithm first retrieves the delta model specification for the provided output port and mode from the ADS. It then concretizes the delta model specification expression with the data from the delta model database. The `delta-model` function sets delta model parameters which are correctable gains to the delta model parameter values from the database. The `delta-model` function sets all other delta model parameters to their respective ideal values, as defined in the ADS. The ADS specifies delta model parameter type and whether the delta model parameter is correctable or uncorrectable.

9.2.5 Analog and Digital Quality Measures

The scaling problem works with analog and digital quality measures. These measures limit how much the scaling procedure can compress any individual analog signal, digital signal, or value. These quality measures work with signal amplitudes. The *amplitude* of a signal with interval $[y, y']$ is the dynamic range $(y' - y)$ of the signal, provided the signal is dynamic $(y' \neq y)$. It is the amplitude of the value $(|y|)$ if the signal is a constant value $(y = y')$. The `amplitude` : $Intervals \rightarrow \mathbb{R}^+$ computes the amplitude of a given interval. I describe the quality measures below:

Analog Quality Measure [AQM]: The AQM is a parameter that specifies the minimum allowed signal-to-noise ratio for the analog signals in the ADP. For constant analog signals, the AQM specifies the lower bound for the ratio of the magnitude of the signal to the analog noise (AQM $\leq$

$|signal|/noise$). If the analog signal is dynamic, the AQM specifies the upper bound for the ratio of the noise to the dynamic range of the signal (AQM $\leq (max(signal) - min(signal))$).

Digital Quality Measure [DQM]: The DQM is a parameter that specifies the minimum allowed signal-to-noise ratio for the digital signals in the ADP. For constant digital signals, the DQM specifies the lower bound for the ratio of the magnitude of the signal to the quantization error (DQM $\leq |signal|/error$). For dynamic digital signals, the DQM specifies the lower bound for the ratio of the dynamic range of the signal to the error (DQM $\leq (max(signal) - min(signal))/error$).

While `LScale` accepts a user-specified minimum AQM and DQM (AQMMIN and DQMMIN), it can also automatically derive the best AQM and DQM for the provided analog device program. It does this by incorporating the AQM and DQM into the objective function.

9.3 CGP Generation

The CGP generation procedure produces a constraint problem that encodes all of the quality, physical realizability, and recovery requirements imposed on the input ADP. The solution to this constraint problem is a valid set of mode selections and scaling transform. The CGP models block mode selections as integer-valued mode selection variables. where each mode selection variable and value corresponds to a block mode. The CGP defines the scaling transform and mode-dependent block properties as positive real variables. The CGP encodes all the quality, physical realizability, and recovery requirements as geometric program constraints. The CGP encodes all the mode-property variable relationships with CGP integer constraints. With this encoding, the CGP can be concretized with a set of mode selections to derive a GP which can be solved with a convex solver to get an optimal scaling transform with respect to some objective function. The scale transform obtained by solving the GP will still respect all the physical limitations and behaviors of the device because these restrictions are encoded as geometric program constraints. This section will formalize the procedure for generating the CGP from the following `LScale` compilation pass inputs:

DSS: The CGP generation procedure uses the DSS interval annotations to compute the unscaled signal ranges for each port in the input unscaled ADP.

ADS: The CGP generation procedure uses the ADS to identify the operating range and frequency limitations and the noise and quantization error for each block which appears in the ADP.

Delta Model Database and Calibration Strategy: The CGP generation procedure uses the delta model database and calibration strategy to compensate for any observed behavioral deviations. Refer to the discussion on factor constraints and the master expression elicitation section (Section 9.4) for more information.

Minimum AQM and DQM: The CGP generation procedure produces a CGP which ensures the scaled circuit AQM and DQM clear the user-defined minimum analog and digital quality measures.

variable	values	type	scope	usage
magnitude scale factor	$\mathbb{R}^+$	scale transform	$(DataFields \cup Ports) \times Locs$	$u(p, l)$
time scale factor	$\mathbb{R}^+$	scale transform		τ
injection variable	$\mathbb{R}^+$	scale transform	$DataFields \times Locs \times \mathbb{I}$	$iv(df, l, k)$
selected mode	$\mathbb{I}$	mode selection	$Blocks \times Locs$	$mv(b, l)$

Table 9.1: Summary of scale transform and mode selection variables.

variable	values	scope	usage	source
process variation	$\mathbb{R}^+$	$OutputPorts \times Locs \times \mathbb{I}$	`dv-prop`(op, l, k)	multi†
operating range	$Intervals$	$Ports \cup DataFields$	`op-range-prop`(p, l)	ADS
maximum frequency	$\mathbb{R}^+$	$Ports$	`max-freq-prop`(p, l)	ADS
analog noise[1]	$\mathbb{R}^+$	$Ports \times Locs$	`noise-prop`(p, l)	ADS
quantization error[2]	$\mathbb{R}^+$	$(Ports \cup DataFields) \times Locs$	`quant-prop`(df, l)	ADS
unscaled interval[3]	$Intervals$	$(Ports \cup DataFields) \times Locs$	`ival-prop`(p, l)	DSS
unscaled amplitude[3]	$\mathbb{R}^+$	$(Ports \cup DataFields) \times Locs$	`ampl-prop`(p, l)	DSS

Table 9.2: Summary of mode-dependent property variables. [1] this property is only specified for analog ports. [2] this property is only specified for digital ports. [3] Not a mode-dependent variable. † The process variation property values are derived from the delta model database, calibration strategy, and ADS.

These quantities ensure none of the analog or digital signals are scaled down to the point where they are overtaken by noise and error.

9.3.1 CGP Variables

The CGP works with a variety of positive real-valued and integer-valued variables. Table 9.1 introduces the CGP variables which make up the scaling transform and mode selections and Table 9.2 presents the CGP mode-dependent properties. The *values* column presents the values each type of variable may take on, the *type* column presents whether the variable is part of the scale transform or mode selections, the *scope* column presents the space of variables of this type, the *usage* column presents an example variable description of that type. The *source* column of the property table presents where the mode-dependent property values are derived from. I summarize the variables below:

Magnitude Scale Variables: The core scaling transform is made up of positive, real-valued per-port/per-data field magnitude scale variables ($u(p, l)$ and $u(df, l)$), which capture the degree to which the magnitude each signal and data field is scaled.

Time Scale Variable: The CGP contains a single time scale variable τ which captures the integration speed of the scaled circuit relative to the baseline hardware integration speed. This variable is mapped to a positive, real value that captures the integration speed relative to the baseline hardware integration speed. If τ is mapped to one, then the integration speed of the scaled circuit is the baseline hardware integration speed, as defined by the hardware time constant.

Injection Variables: The computed scale transform also contains injection variables ($iv(df, l, k)$). The injection variables allow the compiler to modify the function implemented at each expression data field to more flexibly scale the input ADP. The compiler allocates each expression data field df with n arguments a collection of $n+1$ injection variables. Arguments 0..k..n of data field df have injection variables $iv(l, df, j)..iv(l, df, j)..iv(l, df, n)$. The output of the data field has the injection variable $iv(l, df, n+1)$. `LScale` modifies the expression e implemented by the data field to include the injection variables:

$$iv(l, df, n+1) \cdot \texttt{sub}(e, \{(v_0, iv(l, df, 0) \cdot v_0), .., (v_n, iv(l, df, n) \cdot v_n)\}$$

The above function multiplies each occurrence of each argument v_i by its respective injection variable $iv(l, df, i)$. The entire expression is multiplied by the output injection variable $iv(l, df, n+1)$.

Mode Selections: The mode selection variables ($mv(b, l)$) encode the selected mode for each block instance in the ADP. The `mode-int` function maps modes to integer values. The assigned mode selection variable value typically sets the property variable values in the generated CGP. The concretized CGP resolves all mode selection variables to integer values.

Properties: The program also works with port- and data field-specific property variables that encode the operating ranges, maximum frequencies, noise, and quantization error associated with block ports and data fields presented in the analog device specification. All property variables are either mode-invariant (and therefore fixed) or mode-dependent and assigned to constant values once a mode is selected.

The quantization error property variables encode the maximum digital error for each data field and digital port. The noise property variables encode the analog noise associated with each analog port. The operating range and maximum frequency properties encode the operating range and the maximum frequency supported by each port and data field. The operating range property takes on interval values. All other properties take on positive, real-valued values. Note that the constraint generation procedure can translate constraints over intervals to constraints over positive real numbers. Refer to Appendix A.3.1 for more information. The process variation property variables encode the discrepancy between the empirically observed behavior of the block's output port and the expected, idealized behavior. Refer to Section 9.4 for more information on how the change variable values and change variables are generated.

The unscaled interval and amplitude properties encode the intervals and the amplitudes of the unscaled signals in the input ADP. The compiler derives these intervals by propagating the dynamical system variable intervals defined in the DSS through the ADP. Refer to Appendix A.1 for more information on interval propagation. For some port p belonging to block instance (b, loc), the interval and amplitude property values are `signal-ival`(p, l) and $|$`signal-ival`$(p, l)|$ respectively.

Note that these property values are not dependent on the mode.

Process Variation Property Variables: The behavioral deviations from manufacturing variations are incorporated into each output port's dynamics as mode-dependent process variation property variables. Process variation property variables $\texttt{dv-prop}(l, op, i)$ are positive, real-valued variables that capture deviations from the expected behavior at the output port instance (op, l). An output port instance may be associated with one or more process variation property variables. Section 9.4.2 describes how `LScale` derives a single symbolic expression containing process variation property variables that captures all of the behavioral deviations observed at an output port instance relative to a reference expression which captures the expected behavior. Section 9.4 describes how process variation property variables are incorporated into the CGP problem.

Symbolically Applying the Scaling Transform (`apply-xform`)

In many places, `LScale` substitutes in the above magnitude scale variables into an unscaled symbolic expression to derive a scaled symbolic expression. The `apply-xform` function accepts as input an unscaled hardware expression e and the location l of the enclosing block instance and returns the expression with the magnitude scale variables applied. The scaling transform application routine makes the following substitutions:

Variables: All constant data field ($df \in DataFields$ where $\texttt{df-type}(df) = \texttt{constant}$) and port ($p \in ports$) variables are scaled by their respective magnitude scale factors:

$$\texttt{apply-xform}(l, df) = u(df, l) \cdot df \quad \texttt{apply-xform}(l, p) = u(p, l) \cdot p$$

Integration Operations: The derivatives of all integration operations are scaled by the reciprocal of the time scale factor τ^{-1}. This transformation exploits a property of ordinary differential equations (ODEs) to change the integration speed:

$$\texttt{apply-xform}(l, \texttt{integ}(e, e')) = \texttt{integ}(\tau^{-1} \cdot \texttt{apply-xform}(l, e), \texttt{apply-xform}(l, e'))$$

The τ^{-1} term controls the ratio of the scaled derivative to the scaled state variable. A small τ value scales down the derivative of the signal relative to the magnitude of the signal. A large τ value scales up the derivative of the signal relative to the

Expression Data Field Invocations: All expression data field invocations are modified to incorporate the injection variables:

$$\begin{aligned}&\texttt{apply-xform}(l, \texttt{call}(df, [e_0, ..., e_n]) = \\ &iv(df, l, n+1) \cdot \texttt{call}(df, [\quad iv(df, l, 0) \cdot \texttt{apply-xform}(l, e_0), \\ &\qquad\qquad\qquad\qquad iv(df, l, n) \cdot \texttt{apply-xform}(l, e_n)])]\end{aligned}$$

The data field df is an expression data field $\texttt{df-type}(df) = \texttt{expression}$. The above rule multiplies the expression data field result with the injection variable $iv(df, l, n+1)$ and multiplies

each argument e_i with the injection variable $iv(df, l, i)$.

All Other Operations: The scale transform application function recursively traverses all other mathematical operations:

$$\texttt{apply-xform}(l, e \cdot e') \quad = \texttt{apply-xform}(l, e) \cdot \texttt{apply-xform}(l, e')$$

For example, the algorithm recursively applies the scaling transform to the operands of the above multiplication operation and returns the product of the returned sub-expressions.

9.3.2 Combinatorial Geometric Programming Problem Formulation

The combinatorial geometric programming problem CC is a constraint problem composed of geometric programming and SMT constraints. The type of each constraint is written in brackets – each constraint is either a geometric programming constraint **gc** or a CGP integer constraint **int-cstr**.

Block Mode Linkage [int-cstr]: The CGP contains block mode linkage constraints which ensure each mode selection variable $mv(b, l)$ can only be assigned to values which correspond to block modes:

$$\bigvee_{m \in \texttt{modes}b} mv(b, l) = \texttt{mode-int}(m)$$

The above statement limits the mode variable to one of the listed modes for the block b.

Property-Mode Linkage [int-cstr]: The CGP generates property-mode linkage constraints for each mode-dependent property in the CGP. These constraints model the effect of selecting the mode on the physical limitations and behaviors of the target block instance:

$$mv(b, l) = \texttt{mode-int}(m) \implies \texttt{noise-prop}(op, l) = \texttt{noise}(op, m)$$

The above statement assigns the `noise` property for the output port op at location l to y^+ when the mode variable for the block b at location l is set to mode m. I use the $\implies$ operator to model the effect of mode selections on each block instance's property values. The CGP generator produces these constraints for the analog noise, quantization error, operating range, and maximum frequency properties by looking up the property values in the ADS. The master expression elicitation procedure produces the implication constraints for the process variation properties (Section 9.4). The compiler assigns unscaled interval and amplitude properties to constant values and intervals. Note that the compiler resolves all interval properties to positive, non-zero values through additional analysis. Refer to Appendix A.3 for information on the interval encoding tricks utilized by the compiler.

Operating Range Limitations [gc]: For each port p and data field df in the ADP, the CGP ensures the operating range of the port contains the dynamic range of the scaled signal or value.

These operating range constraints take the following form:

$$u(p,l) \cdot \texttt{ival-prop}(p,l) \subseteq \texttt{op-range-prop}(p,l)$$

The above constraint captures the above hardware operating range restriction for a port p. The signal in question is scaled by the magnitude scaling factor $u(p,l)$ and has an unscaled dynamic range of $\texttt{ival-prop}(p,l)$. The operating range of the port is $\texttt{op-range-prop}(p,l)$. Note that the CGP does not directly support $\subseteq$ operations over intervals. The CGP generator performs additional analysis to translate the above constraint into multiple constraints over positive, real numbers. The compiler also adds the same kind of constraint for each data field df:

$$u(df,l) \cdot \texttt{ival-prop}(df,l) \subseteq \texttt{op-range-prop}(df,l)$$

The $u(df,l)$, $\texttt{ival-prop}((,df),l)$, and $\texttt{op-range-prop}(df,l)$ quantities capture the data field magnitude scale factor, unscaled data field value, and mode-dependent data field operating range respectively.

Frequency Limitations [gc]: For each port p with a defined maximum frequency in the ADS, the integration speed of the produced scaled ADP must be lower than the maximum frequency. The following constraint ensures the simulation speed of the scaled ADP does not exceed the maximum supported speed of the device:

$$\tau \cdot tc \leq \texttt{max-freq-prop}(p,l)$$

The integration speed of the scaled computation is the time scaling factor τ times the baseline integration speed of the analog hardware (tc). The integration speed of the scaled computation must not exceed the maximum speed imposed by the port instance ($\texttt{max-freq-prop}(p,l)$).

Analog Quality Restrictions [gc]: The analog quality constraints ensure no signal is scaled down to the point where the signal quality falls below the analog quality measure. For each analog port, the ratio of the amplitude of the scaled signal to the noise must be larger than the analog quality measure (AQM). For each analog port p ($\texttt{sigtype}(p) \in \{\texttt{current}, \texttt{voltage}\}$) in the ADP, `LScale` derives an analog quality restriction constraint of the following form:

$$\text{AQM} \leq \texttt{noise-prop}(p,l)^{-1} \cdot \texttt{ampl-prop}(p,l) \cdot u(p,l)$$

In the above constraint, the ratio of the scaled signal amplitude $u(p,l) \cdot \texttt{ampl-prop}(p,l)$ to the analog noise ($\texttt{noise-prop}(p,l)$) is greater than the analog quality measure AQM.

Digital Quality Restrictions [gc]: The digital quality constraints ensure no digital value or signal is scaled down to the point where the fidelity of the values falls below the digital quality measure. For each digital port or data field, the ratio of scaled signal amplitude to the quantization error must

be greater than than the digital quality measure (DQM). The quantization error is the difference between two consecutive digital decimal values and is automatically computed by dividing the range of the digital port by the number of values. For each digital ADP port p ($\texttt{sigtype}(p) = \texttt{digital}$) and data field df, `LScale` derives a digital quality restriction constraint of the following form:

$$\text{DQM} \leq \quad \texttt{quant-prop}(p,l)^{-1} \cdot \texttt{ampl-prop}(p,l) \cdot u(p,l)$$

$$\text{DQM} \leq \quad \texttt{quant-prop}(df,l)^{-1} \cdot \texttt{ampl-prop}(df,l) \cdot u(df,l)$$

The above digital quality constraints ensure the ratio of the scaled signal amplitude to the digital signal or data field value to the quantization error is greater than the digital quality measure (DQM):

AQM and DQM Restrictions [gc]: The AQM and DQM constraints ensure that the circuit AQM and DQM clear the user-provided minimum AQM and DQM (AQMMIN and DQMMIN).

$$\text{AQM}_{\text{MIN}} \leq \text{AQM} \wedge \text{DQM}_{\text{MIN}} \leq \text{DQM}$$

The above constraint ensures that the scaling transform AQM is larger than the minimum AQM and the scaling transform DQM is larger than the minimum DQM.

External Signal Frequency Restrictions [gc]: For each port p with a defined maximum frequency in the ADS, the speed of each external signal must be lower than the maximum frequency. The following constraint ensures the maximum frequency of the external signals does not exceed the maximum supported speed of the device:

$$\texttt{max-extern-freq} \leq \texttt{max-freq-prop}(p,l)$$

Realtime Execution Speed Restrictions [gc]: If the dynamical system specification defines a realtime minimum and maximum frequency, the integration speed of the produced scaled ADP must fall between the defined minimum and maximum frequency. The following constraint ensures the simulation speed of the scaled ADP falls within the specified frequency range:

$$\tau \cdot tc \leq \texttt{max-realtime-speed}$$

$$\tau \cdot tc \geq \texttt{min-realtime-speed}$$

Recovery Constraints

`LScale` generates CGP constraints that ensure the original simulation dynamics can be recovered from any port within the circuit by scaling the signal magnitude and time by constant factors. These recovery constraints ensure the scaled dynamics of a signal can always be written as the unscaled dynamics of the signal times a constant factor. The recovery constraints also compensate for any behavioral variations observed in the circuit.

Connectivity [gc]: This constraint ensures that the scaling factors of two connected ports are scaled by the same amount. For each ADP connection statement of the form `conn(op,1,ip,1')`,

`LScale` derives a connectivity constraint which ensure the signals at both ports are scaled by the same amount:

$$u(op, l) = u(ip, l')$$

The above constraint ensures the magnitude scaling factor associated with two connected input and output port instances are equal. This constraint ensures the source and destination ports of a signal scale the signal by the same amount.

These connectivity constraints automatically ensure that addition operations implemented via Kirchhoff's law are properly scaled. Take for example two output ports instances ($\langle op, l\rangle$ and $\langle op', l''\rangle$) which are connected to the input port $\langle ip, l'\rangle$. The signal flowing into the input port $\langle ip, l'\rangle$ equals the sum of the signals flowing out of the two output ports. `LScale` will generate two cgp constraints which ensures all three ports are scaled by the same amount:

$$u(ip, l') = u(op, l) \quad u(ip, l') = u(op'', l'')$$

The above constraints ensure it is possible to factor out a constant factor from the sum of the scaled output signals flowing into $u(ip, l')$.

Factor Constraints: The factor constraints ensure that the scaled dynamics of each output port op at location l with all of the associated behavioral deviations is equivalent to the original idealized dynamics scaled by a constant value. To produce factor constraints, the CGP generation algorithm requires a reference mode m_{ref}. The factoring algorithm will use the implemented expression at output port op under mode m_{ref} as the reference expression e_{ref}. This reference expression captures expected dynamics at output port op by the unscaled ADP. The compiler retrieves the reference mode m_{ref} from the modes listed in the block instance's `config` statement:

$$m_{ref} \in M \mid \texttt{config}(\texttt{block}(op), l, M, A)$$

I formally describe the reference mode selection process above. The selected reference mode m_{ref} is chosen from one of the listed modes in the associated block instance's `config` statement. `LScale` first invokes the factoring algorithm to factor out a scaling expression from the block's observed dynamics. The factoring algorithm also derives a set of factoring constraints that must hold for the factor operation to complete successfully:

$$CC, mo = \texttt{factor}(cs, op, l, m_{ref})$$

The above factoring algorithm `factor` factors out a scale expression from the scaled dynamics of the signal at output port instance $\langle op, l\rangle$. It returns the factored out scale expression mo and a

set of CGP constraints CC which must hold for the factoring operation to complete successfully. The monomial mo may contain scale transform variables and process variation variables. All mode-dependent process variation variables capture the deviations in the behavior of the signal relative to the provided reference mode m_{ref}. The factor constraints CC ensure the computed scaling transform preserves the original dynamical system dynamics.

The CGP generation algorithm adds all of the computed constraints CC returned by the factoring algorithm to the CGP. It also adds the following constraint that links the factored scale expression to the magnitude scale factor of the port instance $\langle op, l \rangle$

$$u(op, l) = mo$$

The above constraint ensures the magnitude scale factor assigned to the output port op matches the monomial mo factored out of the output port.

9.4 CGP Factor Constraint Generation

The factor constraints ensure that the scaled dynamics of each output port op with all of the associated correctable behavioral deviations is equivalent to the expected unscaled dynamics of the output port scaled by a constant value. The factor constraint generation algorithm accepts an output port instance $\langle op, l \rangle$ and a reference mode m_{ref} as input. It returns a set of CGP constraints and the factored monomial of scale transform variables (magnitude, time, and injection) and process variation variables.

Algorithm 15 Factor constraint generation algorithm.

1: **function** FACTOR(cs, op, l, m_{ref})
2: **let** e_{ref} = `portexpr`(op, m_{ref})
3: **let** CC, e_{obs} = `master`(cs, op, l, e_{ref})
4: **let** GC, mo, e_{idl} = `fact`(`apply-xform`(l, e_{obs}))
5: **assert** $e_{idl} \equiv e_{ref}$
6: **return** $mo, CC \cup GC$

The algorithm above summarizes the general operation of the factor constraint generation algorithm. It first looks up the reference input-output relation e_{ref} implemented at output port op when the block is placed in the reference mode m_{ref}. It then invokes the master expression elicitation procedure to obtain a single symbolic master expression e_{obs}. The symbolic master expression implements the correctable delta models at output port instance $\langle op, l \rangle$ under calibration strategy cs relative to the reference expression e_{ref}. The master expression e_{obs} implements multiple correctable delta models depending on how the the mode variable $mv($`block`$(op), l)$ is instantiated.

The master expression elicitation procedure also returns a set of CGP constraints that must hold for the master expression to faithfully implement the output port's delta models. These constraints

further restrict the acceptable modes for the associated mode selection variable $mv(\texttt{block}(op), l)$. The master expression implements the reference expression e_{ref} when all the process variation property variables are set to one.

Finally, the factoring algorithm invokes the expression factoring routine `fact` (Section 9.4.1) on the master expression with the scaling transform applied $\texttt{apply-xform}(l, e_{obs})$. The expression factoring routine factors out the scaling transform and process variation property variables from the signal. This routine returns the factored out monomial mo, an unscaled expression e_{idl}, and the set of geometric programming constraints (GC) which must hold for the factoring operation of complete successfully. The returned unscaled expression e_{idl} is equivalent to the reference expression e_{ref} provided to the master expression elicitation procedure. The `factor` routine returns the factored monomial and the combined set of CGP constraints returned by the master expression elicitation and expression factoring routines.

Preservation of Dynamics: The constraint generation algorithm ensures the following relation holds under the generated CGP and GP constraints:

$$\texttt{apply-xform}(l, e_{obs}) = mo \cdot e_{ref}$$

The above equality relation ensures the master expression for the output port instance $\langle op, l \rangle$ with the scaling transform applied equals the output port's reference expression e_{ref} times the scaling expression monomial (mo) derived by the factoring algorithm. This monomial resolves to a positive, constant value when the CGP is solved.

9.4.1 Expression Factoring Algorithm (`fact`)

The `fact` function accepts as input a scaled expression to factor and produces both a set of geometric programming constraints GC, the factored monomial mo, and the unscaled expression e_{idl}. These factor constraints ensure `LScale` can factor out a monomial expression mo made up of scale transform and process variation property variables from the input scaled expression.

The factor function `fact` accepts an input expression e which may contain magnitude scale transform variables (magnitude and time scaling variables and injection variables) and process variation property variables. It produces constraints that make it possible to factor out a monomial made up of scale transform and process variation property variables from the scaled expression. It returns the set of geometric programming constraints GC and the factored monomial mo, and the unscaled expression e_{idl}. This unscaled expression contains no CGP variables. This factoring algorithm upholds the following relation if the returned geometric constraints GC holds:

$$mo \cdot e_{idl} \equiv e$$

The above relation asserts that the original expression is equivalent to the factored monomial times the unscaled expression.

invocation	constraints (GC)	monomial (mo)	unscaled expr (e_{idl})
$\texttt{fact}(y)$		1	y
$\texttt{fact}(p)$		1	p
$\texttt{fact}(df)$		1	df
$\texttt{fact}(\texttt{dv-prop}(op, l, i))$		$\texttt{dv-prop}(op, l, i)$	1
$\texttt{fact}(u(p, l))$		$u(p, l)$	1
$\texttt{fact}(u(df, l))$		$u(l, df)$	1
$\texttt{fact}(\tau)$		τ	1
$\texttt{fact}(iv(df, l, i))$		$iv(df, l, i)$	1
$\texttt{fact}(e \times e')$		$mo \times mo'$	$e_{idl} \times e'_{idl}$
$\texttt{fact}(e \div e')$		mo/mo'	e_{idl}/e'_{idl}
$\texttt{fact}(e + e')$	$mo = mo'$	mo	$e_{idl} + e'_{idl}$
$\texttt{fact}(\texttt{sgn}(e))$		1	$\texttt{sgn}(e_{idl})$
$\texttt{fact}(\texttt{abs}(e))$		mo	$\texttt{abs}(e_{idl})$
$\texttt{fact}(\texttt{min}(e, e'))$	$mo = mo'$	mo	$\texttt{min}(e_{idl}, e'_{idl})$
$\texttt{fact}(\texttt{max}(e, e'))$	$mo = mo'$	mo	$\texttt{max}(e_{idl}, e'_{idl})$
$\texttt{fact}(\texttt{exp}(e))$	$mo = 1$	1	$\texttt{exp}(e_{idl})$
$\texttt{fact}(\texttt{ln}(e))$	$mo = 1$	1	$\texttt{ln}(e_{idl})$
$\texttt{fact}(\texttt{sin}(e))$	$mo = 1$	1	$\texttt{sin}(e_{idl})$
$\texttt{fact}(\texttt{cos}(e))$	$mo = 1$	1	$\texttt{cos}(e_{idl})$
$\texttt{fact}(\texttt{pow}(e, y'))$	$mo' = 1$	$mo^{y'}$	$\texttt{pow}(e_{idl}, y')$
$\texttt{fact}(\texttt{pow}(e, e'))$	$mo = mo' = 1$	1	$\texttt{pow}(e_{idl}, e'_{idl})$
$\texttt{fact}(\texttt{integ}(e, e'))$	$mo' = mo$	mo'	$\texttt{integ}(e_{idl}, e'_{idl})$
$\texttt{fact}(\texttt{call}(df, [e_0, ..e_j.., e_n]))$	$mo_j = 1$	1	$\texttt{call}(df, [e_0, ..e_j.., e_n])$
$\texttt{fact}(\texttt{emit}(e))$		mo	$\texttt{emit}(e_{idl})$
$\texttt{fact}(\texttt{extvar}(e))$		mo	$\texttt{extvar}(e_{idl})$

where $\langle GC, mo, e_{idl}\rangle = \texttt{fact}(e)$, $\langle GC', mo', e_{idl}\rangle = \texttt{fact}(e')$, $\langle GC_j, mo_j, e_{j,idl}\rangle = \texttt{fact}(e_j)$

Figure 9-2: General operation of the `fact` function.

Figure 9-2 presents the operation of the `fact`. the leftmost column presents the `fact` invocation. Columns 2-4 present the geometric program constraints which are added, the returned monomial, and unscaled expression respectively. Each row adds the necessary constraints to ensure the above relation holds. I summarize its operation below:

- $\texttt{fact}(y)$: The constant value y cannot be scaled, and therefore has the scaling factor 1. The unscaled expression is y.
- $\texttt{fact}(\texttt{dv-prop}(op, l, i))$: The process variation property variable $\texttt{dv-prop}(op, l, i)$ scales the constant value `1` by the monomial $\texttt{dv-prop}(op, l, i)$. `LScale` must factor out or eliminate all process variation property variables from the block dynamics to ensure all behavioral variations can be corrected for by scaling the circuit.
- $\texttt{fact}(p)$,$\texttt{fact}(df)$: All port and data field variables are scaled by `1`. In this formulation, the magnitude scale variables are already incorporated into the target scaled expression.
- $\texttt{fact}(u(p, l))$,$\texttt{fact}(u(df, l))$,$\texttt{fact}(\tau)$, $\texttt{fact}(iv(df, l, k)$: All magnitude and time scale variables

scale a constant value `1` by the monomial $u(l,p)$, $u(l,df)$, and τ respectively. All injection variables scale the constant value `1` by the monomial $iv(df,l,k)$.

For the remaining invocations, I introduce sub-expressions e and e' which implement $mo \cdot e_{idl}$ and $mo \cdot e'_{idl}$ provided constraints GC and GC' hold. The operation of the `fact` algorithm on these rules is summarized below:

- *`fact`*$(e \cdot e')$ *and* *`fact`*(e/e'): The monomial $mo \cdot mo'$ can be factored out of the scaled expression $mo \cdot e_{idl} \cdot mo' \cdot e'_{idl}$. The monomial mo/mo' can be factored out of the scaled expression $mo \cdot e_{idl}/(mo' \cdot e'_{idl})$. Both factoring operations are achievable by simply shuffling around the terms.

- *`fact`*$(e + e')$ *and* *`fact`*$(e - e')$: The monomial mo can be factored out of a scaled expression $mo \cdot e_{idl} + mo' \cdot e'_{idl}$ or $mo \cdot e_{idl} - mo' \cdot e'_{idl}$ provided $mo = mo'$. The resulting factored expressions implement $mo \cdot (e_{idl} + e_{idl})$ and $mo \cdot (e_{idl} - e_{idl})$ respectively.

- *`fact`*$($*`pow`*$(e, e'))$: If e' evaluates to a constant value y, the scaling factor mo^y can be factored out of the scaled expression `pow`$(mo \cdot e, mo' \cdot x)$ provided $mo' = 1$. If e is not a constant value, the expression cannot be scaled and therefore has a scaling factor of 1. For the scaling factor to be 1, mo must also be 1.

- *`fact`*$($*`integ`*$(e, e'))$: The scaling factor mo can be factored out of the integration operation $\int mo \cdot edt$ with an initial value of $mo' \cdot e'$ provided the factored monomial of the initial condition mo' equals the factored monomial of the integrated signal mo. The factoring algorithm adds a constraint $mo = mo'$ that ensures the initial value and the time-varying values of the integrated signal are scaled by the same amount.

- *`fact`*$($*`call`*$(df, [e_0, .., e_j, .., e_n]))$: For a call operation that invokes the expression stored in expression data field df (`df-type`(df) = `expression`) with arguments $[e_0..e_n]$, The factoring algorithm ensures the each factored monomial associated with an input argument equals 1. It therefore adds a constraint of the form $mo_j = 1$ for each input expression e_j. Typically, the factored monomials for these input arguments have injection variables which provide additional degrees of freedom for meeting this constraint. These variables become part of the expression assigned to the data field df.

9.4.2 Master Expression Elicitation (`master`)

The master expression elicitation procedure (`master`) derives a master expression that implements the correctable delta model expressions for an output port instance relative to a reference expression e_{ref}. The master expression elicitation procedure introduces mode-dependent process variation

property variables when necessary to harmonize mode-dependent disparate behaviors across delta models. Each process variation property variable `dv-prop`(l, op, i) is uniquely identified by the output port instance (op, l) of the associated master expression and its numerical identifier $i \in \mathbb{I}$. The derived master expression has the following properties:

- Setting all the process variation property variables to one transforms the master expression to the reference expression.
- There is a set of process variation property variable assignments for each block mode that transforms the master expression to the delta expression for that mode. If there is no such set of assignments for a particular mode, then the master expression elicitation procedure returns constraints that explicitly forbid the selection of the offending mode.

Algorithm 16 Master expression elicitation (`master`).

```
# cs: Target calibration strategy.
# op: Output port identifier of target output port instance.
# l: Location of target output port instance.
# e_ref: Reference expression.
function MASTER(cs,op,l,e_ref)
    let ME = {delta-model(cs, op, l, m) | m ∈ modes(block(op))}
    let M, CC, e_mst = harm(e_ref, ME)
    if M ≠ ∅ then
        return CC, e_mst
    else
        error harmonization failed
```

Master Expression Elicitation: Algorithm 16 presents the master expression elicitation procedure. The elicitation procedure accepts as input a calibration strategy cs, an output port instance (op, l) and a reference expression e_{ref}.

The elicitation algorithm first retrieves the set of correctable delta models for the output port instance $\langle op, l \rangle$ with calibration strategy cs with the `delta-model` delta model retrieval function introduced in Section 9.2. The delta model retrieval function only allows delta model parameters that implement correctable gains to deviate from their ideal values in the returned delta model. The delta model database is repeatedly queried with each block mode to build a set of mode-expression tuples $ME \in \mathbb{P}(Modes \times Exprs)$ which capture the correctable behavioral deviations of output port instance under each mode. I will refer to ME as the delta mode-expressions for the rest of this section.

The master expression elicitation algorithm then invokes the `harm` function to derive a single symbolic master expression e_{mst} which encompasses both the reference expression behavior and all the behaviors of the delta mode-expressions. The `harm` also returns a set of mode-limiting CGP constraints which must hold for the harmonization operation to succeed and the subset of modes M

which are modeled with the returned master expression. If the returned master expression models zero modes, then the harmonization procedure fails with an error. If the master expression e_{mst} captures at least one mode, it returns the calculated CGP constraints CC and the master expression e_{mst}.

Harmonization Function (`harm`)

The harmonization function identifies a unifying master expression that captures both the reference expression's behavior e_{ref} and the behaviors of all of the delta mode-expressions ME.

Algorithm 17 Harmonization function (`harm`)

```
1: # op: output port identifier of target output port instance.
2: # l: location of target output port instance.
3: # e_ref: Reference expression to harmonize against.
4: # ME: Set of delta mode-expressions to harmonize
5: # returns the set of supported modes, a set of new CGP constraints, and the master expression.
6: function HARM(op, l, e_ref, ME)
7:     let M, CC, e_mst = harm-direct(op, l, e_ref, ME)
8:     if sameOp(e_ref, ME) and M = then
9:         let op = getOp(e_ref, ME)
10:        let CC_decomp = ∅
11:        let M_decomp = {m | ⟨m, e⟩ ∈ ME}
12:        let subexprs = array(getNumArgs(op))
13:        for i in 0...getNumArgs(op) do
14:            let e_i,ref = getArg(e_ref,i)
15:            let ME_i = {⟨m, getArg(e, i)⟩ | ⟨m, e⟩ ∈ ME}
16:            let M_i, CC_i, e_i,mst = harm(op, l, e'_ref, ME')
17:            let M_decomp = M_decomp ∩ M'
18:            let CC_decomp = CC_decomp ∪ CC'
19:            let subexprs[i] = e_mst
20:        let e'_mst = buildExpr(op,subexprs)
21:        return M_decomp, CC, e'_mst
22:    else
23:        return M, CC, e_mst
```

Algorithm 17 presents the harmonization algorithm. The algorithm accepts as input the output port instance (op, l), the reference expression e_{ref}, and the delta mode-expressions ME. It returns the identified master expression e_{mst}, the set of derived of CGP constraints (CC), and the subset of modes captured by the master expression (M).

The harmonization algorithm first invokes the `harm-direct` function to directly harmonize the reference expression with all of the delta model expressions. The `harm-direct` function tries to create a master expression that models the provided reference expression and delta model expressions. If the direct harmonization operation failed (the set of returned modes is empty), then the harmonization algorithm decomposes all the expressions and recursively harmonizes the sub-expressions.

Before decomposing all of the expressions, the algorithm first tests to see if the reference expression and all of the delta-model expressions all have the same root operator (`sameOp`). If all the expressions implement the same operator, the algorithm gets the root operator (`op`)and sets up the initial state of the decomposition procedure. The initial set of captured modes consists of all the modes in the provided delta mode-expression set. The initial set of constraints is the empty set. The algorithm also creates an empty `subexprs` array to later store the computed master sub-expressions.

The algorithm next iterates over each operand accepted by the operator `op`. For each operand with some position `i`, the algorithm identifies the sub-expression at position `i` in the reference expression ($e_{i,ref}$). It also constructs a new delta mode-sub-expression set ME_i which pairs each mode with the ith argument of the associated delta expression. It then invokes `harm` on the reference sub-expression and the delta mode-subexpression set to derive a master subexpression for operator argument `i`. The returned set of captured modes M_i, CGP constraints CC_i, and master expression $e_{i,mst}$ are then used to update the subexpression array (`subexpr`) and the running set of captured modes (M_{decomp}) and CGP constraints (CC_{decomp}).

After all of the sub-expressions have been harmonized, the algorithm builds the final master expression e'_{mst} from the list of master sub-expressions `subexprs` and the root operator `op`. It returns the set of captured modes and CGP constraints (M_{decomp} and CC_{decomp} and the finalized master expression e'_{mst}.

Direct Harmonization (harm-direct)

Algorithm 18 Direct expression harmonization algorithm

1: # **op**: Output port identifier of target output port instance.
2: # **l**: Location of target output port instance.
3: # e_{ref}: Reference expression to harmonize against.
4: # **ME**: Delta mode-expression set to harmonize.
5: # **returns** a set of new CGP constraints and the master expression.
6: **function** HARM-DIRECT(op,l,e_{ref}, ME)
7: **let** $y^{+}_{ref}, e_{base} =$ factor-coeff(e_{ref})
8: **let** $b =$ block(op)
9: **let** $CC = \emptyset$
10: **let** $M = \emptyset$
11: **new** dv-prop(op, l, i)
12: **let** $e_{mst} =$ dv-prop$(op, l, i) \cdot e_{ref}$
13: **for** $\langle m, e \rangle$ **in** ME **do**
14: $y^{+}_{dev}, e_{dev} =$ factor-coeff(e)
15: **if** $e_{dev} \equiv e_{base}$ **then**
16: **let** $cc = \{\{mv(b, l) =$ mode-int$(m) \Rightarrow$ dv-prop$(op, l, i) = y^{+}_{dev}/y^{+}_{ref}\}\}$
17: **let** $M = M \cup \{m\}$
18: **let** CC $= CC \cup \{cc\}$
19: **else**
20: **let** $cc = \{\{mv(b, l) \neq$ mode-int$(m)\}\}$
21: **let** CC $= CC \cup \{cc\}$
22: **return** CC, e_{mst}

Algorithm 18 presents the direct harmonization function harm-direct(op, l, e_{ref}, ME). This algorithm harmonizes the provided reference expression e_{ref} with a set of delta mode-expressions ME. It returns the resulting harmonized master expression, the computed CGP constraints, and the set of modes captured in the master expression. The computed master expression must equal the reference expression when all the process variation property variables are one. The computed master expression must also resolve to the appropriate mode-dependent delta expression when the mode selection variable $mv($block$(op), l)$ is set to a specific mode. The direct harmonization function produces a master expression that fulfills the above conditions.

The harm-direct function harmonizes a reference expression e_{ref} with a set of delta mode-expressions ME. It first decomposes the reference expression e_{ref} into a positive constant coefficient y^{+}_{ref} and basic expression e_{base} where $y^{+}_{ref} \cdot e_{base} = e_{ref}$. The factor-coeff utility function factors out a positive coefficient from the input expression and decomposed expression. Next, the algorithm sets up the starting set of constraints CC and modes M captured by the derived master expression – both sets are initially empty. The direct harmonization algorithm then instantiates a new process variation property variable dv-prop(op, l, i) and defines the master expression for the direct harmonization procedure:

$$e_{mst} = \text{dv-prop}(op, l, i) \cdot e_{ref}$$

The above master expression equals the reference expression when the process variation property variable dv-prop(op, l, i) is set to one. Next, the algorithm must ensure the master expression implements the correctable delta model for each mode. The algorithm iterates over each mode-expression pair in the delta mode-expression set and then attempts to implement each delta expression with the master expression. It rewrites each delta expression as the product of a positive constant coefficient y^+_{dev} and a base expression e_{dev}. If the base deviation expression is equivalent to the base reference expression, then the algorithm adds the appropriate process variation property variable linkage constraints. In that case, the algorithm can transform the master expression to the delta expression of the current mode by setting the process variation property variable dv-prop(op, l, i) to a value. The algorithm introduces a new implication CGP constraint that instantiates the process variation property variable to the y^+_{dev}/y^+_{ref} value:

$$\{\{mv(b, l) = \texttt{mode-int}(m) \Rightarrow \texttt{dv-prop}(op, l, i) = y^+_{dev}/y^+_{ref}\}\}$$

The assigned property value computes the ratio of the coefficient of the delta expression to the coefficient of the reference expression. When this mode is selected, the master expression implements the following expression:

$$y^+_{dev}/y^+_{ref} \cdot y^+_{ref} \cdot e_{base}$$

This expression simplifies to $y^+_{dev} \cdot e_{base}$ – the delta expression for the current mode. If the two base expressions are not equivalent $e_{base} \not\equiv e_{dev}$, then the master expression cannot implement the delta expression for that mode. The algorithm adds a CGP constraint that ensures the mode selection variable for the offending block instance is not assigned to the offending mode:

$$\{\{mv(b, l) \neq \texttt{mode-int}(m)\}\}$$

The above constraint ensures the mode of the enclosing block instance $mv(b, loc)$ is not assigned to the offending mode m. After all of the mode-expression tuples have been processed, the algorithm returns the collected CGP constraints and computed master expression.

9.5 Completing the ADP

The compiler first generates and solves the CGP to obtain a set of mode selections. The compiler then applies the mode selections to the ADP to complete the ADP. The compiler also applies the mode selections to the CGP to derive a GP which computes the scaling transform. The compiler then solves the GP to minimize the provided scaling objective function. The compiler applies the derived scaling transform to the completed ADP.

9.5.1 Mode Selection

`LScale` first nondeterministically produces a set of mode assignments by solving the CGP with an SMT solver. With this formulation, `LScale` is always able to find valid mode assignments if they exist. If the CGP is infeasible, then the underlying GP is also infeasible for all possible mode selections. In this situation, the ADP is unscalable and not executable on the analog hardware. `LScale` finds alternate mode selections by progressively blacklisting mode selection variable assignments that it has already generated. Note that the SMT problem is often solved easily, as the subset of SMT constraints I use is fairly restrictive.

9.5.2 Scale Transform Generation

The compiler substitutes all the derived mode selections into the CGP – this resolves all property variables to values and eliminates all of the SMT constraints and discrete variables from the constraint problem. This operation transforms the CGP into a pure geometric programming problem which can be optimized using a convex solver. It pairs the derived geometric programming problem with the user-defined scaling objective function po_{opt}. The scaling objective function defines the criteria to minimize when scaling the circuit. The convex solver finds the scaling transform comprised of magnitude and time scale factors and injection variable values that minimizes the provided scaling objective function. Currently, the compiler supports scaling objective functions which minimize execution time and maximize analog and digital signal quality for the target ADP:

- **Maximum Speed**: The objective function $po_{opt} = \tau^{-1}$ maximizes the simulation speed. This objective function is useful for minimizing the simulation time and enabling developers to understand the range of feasible simulation times.

- **Maximum Quality**: The objective function $po_{opt} = \text{AQM}^{-1} \cdot \text{DQM}^{-1}$ maximizes the analog and digital quality measures for the circuit. This objective function is useful for producing scaled circuits which maximize the quality of the computation.

- **Balanced Quality:** The objective function $po_{opt} = \tau^{-1} \cdot \text{AQM}^{-1} \cdot \text{DQM}^{-1}$ maximizes both the quality and the speed of the simulation. This objective function is useful for producing performant scaling transforms that produce high fidelity results. The compiler typically uses the balanced scaling objective function to scale the provided circuits.

This range of implemented objective functions highlights the flexibility of formulating the configuration scaling problem as a geometric program.

9.5.3 Generating the Scaled ADP

`LScale` then incorporates the set of mode selections and the scaling transform to the ADP. Given an ADP configuration for block instance $\langle b, l \rangle$, `LScale` makes the following modifications:

1. It assigns a magnitude scale factor to each port and data field in the block instance. The compiler encodes each scale factor assignment as a `scale` statement in the ADP.
2. It assigns the time scale factor to the ADP. The compiler encodes the time scale factor assignment as a `timescale` statement in the ADP.
3. It modifies the set of viable modes to list only the mode assigned to the block instance's associated mode variable $mv(b, l)$. The compiler modifies the `modes` statement in each block configuration to reference the selected mode.
4. It modifies each expression data field assignment to incorporate the computed injection variables. The compiler modifies the `set` statements in the block instance configurations to reference the updated expression data field values.

9.5.4 Implementation

The `LScale` pass uses the Z3 SMT solver to solve the CGP [32]. The `LScale` pass makes use of the `minimize` feature provided by recent versions of the Z3 solver to identify mode selections that minimize the scale objective value. The `LScale` pass also supports invoking the Z3 solver without the `minimize` feature and then using a convex optimizer to solve the scaling problem. The compiler uses this approach of the Z3 solver fails to return an optimal result. In practice, the Z3 solver can consistently identify the optimal result. The `LScale` pass produces multiple scaled ADPs from the same unscaled ADP by progressively blacklisting mode selections that have already been reported.

The `LScale` pass supports finer grain quality measures in practice. It maintains analog quality measures for the state variables and variables externally observed by the user and dedicated digital quality measures for time-varying digital signals. The scaling objective function jointly maximizes all of these quality measures to maximize the quality of the signals and values.

9.6 Conclusion

In this chapter, I rigorously describe the circuit scaling pass of the compiler. Analog devices are subject to a variety of physical phenomena such as noise, quantization error, process variations, and frequency limitations. The compiler scales the computation to respect all of the imposed voltage, current, and frequency limitations and compensate for behavioral variations within the device while delivering acceptably accurate computations in the presence of noise and quantization error. This

scaled computation ensures the original dynamical system dynamics are recoverable at runtime by applying a compiler-derived inverting transform. When scaling the circuit, the circuit scaling pass may selectively change the block modes in the analog circuit to better scale the circuit.

I first introduce the geometric programming (GP) and combinatorial geometric programming (CGP) constraint problems that I use throughout this chapter. The compiler formulates the circuit scaling problem as a CGP, a type of mixed-integer constraint problem which contains both positive non-zero real-valued variables and integer-valued variables. The compiler solves the CGP with an SMT solver. A key property of the CGP is that it reduces to a geometric programming problem when all of the integer-valued variables are assigned to values. The geometric programming problem is a type of convex optimization problem which supports non-linear constraints and objective functions. The solution to the geometric programming problem returned by the convex solver is guaranteed to be optimal – that is, it minimizes the provided objective function.

I then provide an overview of the overall operation of the circuit scaling procedure. The circuit scaling procedure first derives a CGP that captures the physical limitations and behaviors present in the hardware. The CGP generation step accepts as input the unscaled ADP, the ADS and DSS, and the delta model database and the calibration strategy to target. The delta model database, delta model specifications, and calibration strategy together define the delta models for the device on hand. The CGP generation step also optionally accepts minimum analog and digital quality measures that limit the extent to which the scaling transform can scale down any single signal or value. The CGP variables encode the scaling transform and the set of selected block modes. The CGP is then solved to produce a set of mode modifications which are then applied to the CGP to produce a GP. The GP is then paired with a scaling objective function which captures the circuit criteria to optimize. The GP is solved with a convex solver to produce an optimal scaling transform that minimizes the provided scaling objective function. The compiler applies both the scaling transform and the mode modifications to the unscaled ADP to produce a scaled ADP.

I then introduce the notation for formally specifying the scaling problem. I extend the ADS notation presented in Chapter 8 to include operating range, frequency, noise, quantization error, and delta model specification information. I extend the DSS notation presented in Chapter 8 to include interval annotations. I extend the ADP notation from Chapter 8 to include notations for describing the calibration strategy and delta models. I also formally define the user-provided analog and digital quality measures.

I next formally introduce the CGP generation procedure. I first described how the scaling transform, mode selections, and ADP properties are captured in the CGP. I introduce the concept of an injection variable – a constant coefficient that is injected into the expression assigned to an expression data field to more flexibly scale the circuit. I then describe how the compiler generates each of the CGP constraints from the provided inputs. The compiler generates CGP constraints that

link block modes to ADP property values, encode operating range and frequency limitations, capture the effect of quantization error and noise on signal quality, enforce the user-provided minimum quality measures, and ensure real-time computations execute at the correct speed.

The CGP also contains a number of recovery constraints that ensure the original dynamical system dynamics can be recovered from any port in the scaled ADP by applying an inverting transform. Specifically, these constraints ensure the scaled dynamics of a signal with the delta models applied equals the unscaled dynamics of the signal times a constant coefficient. These factor constraints are formulated so the compiler can more freely change the modes of the blocks and more accurately target the device on hand.

The compiler derives these constraints by analyzing the scaled dynamics of the signals at each output port. The compiler derives the scaled dynamics of each signal by symbolically applying the scaling transform to a master expression for that signal. The master expression is a single symbolic expression that captures the behavior of the scaled signal under different modes relative to the ideal behavior of the signal from the unscaled ADP. The circuit scaling procedure incorporates the delta model information into the master expression so that the resulting scaling transform can compensate for any correctable gains present in the device on hand. The master expression also models the mode-dependent behavior of the output port. The mode-dependent property variables in the symbolic expression capture the effect of changing the mode on the master expression. The master expression enables the compiler to adjust the block modes more freely and compensate for behavioral deviations in the hardware.

Further Reading: Refer to Chapter 6 for an introduction to scaled ADPs and the scaling transform. Refer to Chapter 5 for an introduction to delta models, delta model specifications, and the delta model database.

Chapter 10

Results

This chapter presents the empirical evaluation of the compiler presented in this thesis. In this evaluation, I compile twelve benchmarks from the biology, controls, and physics domains (Sections 4.1-Sections 4.12) to the HCDCv2 analog device presented in Chapter 5. This evaluation covers the following concepts:

- **Performance** - I investigate the energy and power consumption and the execution times of the lowest error circuits. I found the lowest error circuits execute in 0.25-1.92 ms, consume 0.10-5.09 μJ of energy and 0.20-1.10 mW of power. The produced waveforms are indistinguishable from the expected dynamics.
- **Compiler Design** - I investigate the impact of different compiler optimizations on the end-to-end result. I study the importance of the scaling transform and the mode selection and delta model compensation optimizations employed by the circuit scaling pass. I find that all investigated aspects of the compiler design are integral to consistently obtaining good quality results.

  I also compare the co-designed calibration strategy to the traditional calibration strategy. Refer to Chapter 5 for an overview of both calibration strategies. I find that for 10 of the 12 benchmark applications, the co-designed calibration strategy produces lower error results or more consistently produces results of the same quality. From this analysis, I can conclude that the co-designed strategy calibrates the device to behave more predictably and eliminates more unwanted behaviors than the traditional strategy.
- **Circuit Optimality** - I investigate the optimality of the produced circuits. The produced circuits use few extraneous analog blocks and often attain the maximum execution speed supported by the device. I also investigate how effectively the signals in the circuits utilize the operating ranges available on the device. While the compiler does not maximize the

signals' dynamic ranges on all wires, it does use a good fraction of the available operating range for a majority of signals.

I also investigate the optimality of the `balanced` scaling objective function values for the produced circuits. I find that the compiler produces a variety of equally viable scaling transforms which all scale the circuit differently but successfully minimize the `balanced` scaling objective function.

- **Unscaled ADP Viability**: I investigate why the produced unscaled circuits cannot be executed on the target hardware platform. I find that all ADPs generated by the circuit synthesis procedure violate at least one operating range or frequency constraint.
- **Circuit Attributes and Quality** - I investigate the relationship between different circuit attributes and the end-to-end result quality. I find that the block instance selection has a significant impact on the quality of the end-to-end result. I am also able to confirm that the `balanced` scale objective function is a good predictor of the end-to-end result quality.
- **Beyond the `balanced` Scaling Objective** - I investigate whether there is potentially a better scaling objective than the `balanced` scaling objective. I compile the benchmark applications with an alternative objective function that maximizes one signal in the target circuit. I find that this alternative objective function produces higher fidelity or better performing, lower energy circuits for a subset of benchmark applications.
- **Real-time Signal Processing Case Studies** - I present two realtime signal processing case studies which directly perform computation on an externally provided, continuously evolving signal.

I next present a high-level roadmap of the results chapter. I summarize the questions addressed in each section and provide an overview of the key results from my evaluation.

Key Results

The compiler presented in this thesis produces scaled ADPs which execute the energy efficiently and performantly on the analog hardware (Table 10.2) with great accuracy (Figure 10-1).

The circuit scaling procedure and all of the circuit scaling optimizations are integral to generating ADPs that produce high quality results. Without the scaling transform, the compiler cannot produce ADPs which can be executed on the analog hardware (Sections 10.3.1 and 10.6). Without mode selection, the compiler produces ADPs that yield poor results for a significant fraction of the benchmarks (10.3.2). The compiler cannot produce executable scaled ADPs for two benchmarks without mode selection. The delta model compensation optimization employed by the circuit scaling

procedure further improves the fidelity of the produced waveforms (Section 10.3.3). Without delta model compensation, the produced waveforms noticeably deviate from the reference signal.

The circuit synthesis procedure produces ADPs that use no additional copier blocks and few additional routing blocks (Sections 10.5.2). For a significant fraction of benchmarks, the compiler incorporates additional multiplier blocks into the ADP to compensate for coefficients introduced by the HCDCv2 blocks.

Experimental Setup (Section 10.1)

I evaluated the compilation toolchain on the twelve benchmark dynamical system applications introduced in Chapter 3. I configure the toolchain to target the HCDCv2 re-configurable device introduced in Chapter 5. For each of the benchmark applications, I compiled the application's DSS with the compiler to produce a collection of two hundred scaled ADPs that all implement the starting DSS. Each of these scaled ADPs uses different block instances, selects different block modes, or implements a different scaling transform. I produce many candidate scaled ADPs for each application so that I may study the effect of scaled ADP characteristics on the end-to-end result. I am also able to study the distribution of compilation outcomes with this setup. Chapter 6 presents a representative ADP for each benchmark application.

- **Compilation (Section 10.1.1)**: The compiler is provided with the DSS of the benchmark application and the ADS for the HCDCv2 analog hardware. The compiler produces 10 distinct unscaled ADPs per application and 20 distinct scaled ADPs for each unscaled ADP. Of the 20 scaled ADPs, the compiler produces 10 scaled ADPs which target blocks calibrated with the `minimize_error` HCDCv2 calibration strategy and 10 scaled ADPs which target blocks calibrated with the `maximize_fit` calibration strategy. Both calibration strategies are described in Chapter 5.

  The compiler produces scaling transforms by minimizing the `balanced` scaling objective function introduced in Chapter 9. The `balanced` scaling objective function maximizes both the analog and digital signal qualities and the execution speed of the scaled circuit. The circuit scaling procedure implements two key optimizations: delta model compensation and mode selection. The delta model compensation optimization allows the compiler to produce scaled ADPs that compensate for the low-level physical behaviors captured by the block delta models. Refer to Chapter 5 for an overview of delta models. The mode selection optimization allows the compiler to change the block modes when producing the scaled ADPs to better scale the circuit.

- **Execution (Section 10.1.2)**: Each compiled scaled ADP is then executed on the Sendyne development board, which interfaces with the HCDCv2 chip [132, 61, 51]. I compare each

recorded signal with a reference waveform computed with a high-precision digital differential equation simulator. I quantitatively capture the degree of agreement between the two signals with the percent normalized root-mean-squared error (% rmse). This quality metric computes the root-mean-squared error between the two signals and then normalizes the measure with the maximum amplitude of the reference signal.

- **Evaluation Metrics (Section 10.1.3)**: The analyses in this chapter often study the distribution of measures across scaled ADPs. To reduce the effect of outlier executions, I use the median, inter-quartile range (IQR), and first and third quartiles to capture the distribution of a particular metric.

Quality, Runtime, Power, and Energy (Section 10.2)

I investigate the power, energy consumption, and execution time for the lowest-error (best-performing) executions for each benchmark application. Section 10.2 presents a quantitative study of the performance characteristics of the lowest-error executions. The lowest-error benchmark applications execute in 0.25-1.92 milliseconds and consume 0.10-5.09 μJ of energy while accruing between 3.461×10^{-6} and 1.96% normalized root-mean-squared error. Section 10.2.1 presents a qualitative study of the degree of agreement between the measured signals and the reference waveforms. Visually, the reference waveforms and recorded signals are indistinguishable from one another. Because the waveforms align, the compile-time estimate of the execution time accurately predicts the actual time required to simulate the dynamical system on the analog hardware.

Evaluation of Compiler Design (Section 10.3)

I investigate to what extent the compiler optimizations enable the compiler to produce circuits consistently that yield high quality results. To better understand the variations across result outcomes, I evaluate the distribution of % root-mean-squared errors across all executions for each benchmark. This analysis investigates the importance of the scaling transform, mode selection, delta model compensation, and calibration strategy on the end-to-end result:

- **Scaling Transform** (Section 10.3.1): I produced executions with circuit scaling turned off. Without circuit scaling, all of the compiled benchmark applications violated hardware constraints and could be executed on the HCDCv2. The scaling transform is therefore integral for producing circuits that can be executed on the analog device.

- **Mode Selection** (Section 10.3.2): I next produced executions with circuit scaling turned on, but master expressions turned off – this prevents the compiler from aggressively changing the mode when scaling the circuit. For six benchmarks, the best-performing executions without

mode selection deviated significantly from the reference waveforms. Two compiled benchmark applications violate hardware constraints and cannot be executed on the HCDCv2. The four benchmarks which produced good results still had higher errors than the best-performing executions from Section 10.2 and did not confer any significant runtime, power, or energy benefits. The mode selection optimization is therefore integral for obtaining good quality results.

- **Delta Model Compensation**(Section 10.3.3): I next compare the executions with and without delta model compensation. Eleven of the benchmarks produce lower error results more consistently with delta model compensation enabled. The inclusion of delta model compensation is therefore helpful in further improving the accuracy of the produced executions.
- **Calibration Strategy** (Section 10.3.4): I next compare the efficacy of the co-designed (`maximize_fit`) calibration strategy with the traditional calibration strategy (`minimize_error`). I found the co-designed calibration strategy can more consistently produce lower error results than the traditional strategy. The co-designed strategy either more consistently delivers comparable error to the traditional strategy or more consistently produces lower error results than the traditional strategy for nine benchmarks. The co-optimized calibration strategy, therefore, helps improve the predictability of the HCDCv2 in many cases. In some situations, the co-optimized calibration strategy is even able to deliver better results.

Compilation Times (Section 10.4)

I then investigate the time required for the compiler to produce the scaled ADPs. This analysis reports the compilation times and provides a performance breakdown of the compiler by compilation pass. The compiler takes between 0.33-34.87 seconds to compile the benchmark applications. For most applications, this time is spent in the `LGraph` pass synthesizing the circuit.

Optimality of Compilation Outcomes (Section 10.5)

I then study the optimality of the unscaled and scaled ADPs produced by the compiler. Section 10.5.1 presents the evaluation metrics which I use in these analyses. I investigate the optimality of the following ADP characteristics in this section:

- **Resource Utilization** (Section 10.5.2): I first investigate the resource utilization of the produced ADPs. Here, an optimal circuit would not incorporate any unnecessary blocks into the ADP. I validate that, for 11 of the 12 applications, the produced ADPs use the minimum number of routing and assembly blocks to implement the target dynamical system. The remaining benchmark uses the minimum number of assembly blocks and two more routing blocks than necessary. The compiler also uses the minimum number of required compute blocks for all compute blocks except the multiplier block. The multiplier blocks are generally used to correct for coefficients introduced by the device. The compiler can therefore produce circuits that implement the desired computations without using too many extraneous blocks.

- **Execution Speed** (Section 10.5.3): I investigate whether the scaled ADPs execute at the maximum speed supported by the HCDCv2. For 7 of 12 applications, the compiler produces at least one scaled ADP which attains the maximum possible speed supported by the device. For 3 of the 12 applications, all scaled ADPs operate at the maximum possible execution speed supported by the HCDCv2. The execution speed is at least 49.7% of the maximum possible execution speed supported by the device for all benchmarks. This analysis demonstrates that, for a majority of applications, the compiler can fully exploit the performance characteristics of the HCDCv2.

- **Dynamic Ranges of Signals** (Section 10.5.4):I next investigate how effectively the scaled ADPs scale the signals within the circuit. It is often desirable to scale signals to have large dynamic ranges since signals with large dynamic ranges are more robust to noise and quantization error. It is generally not possible to maximize the dynamic range of all the signals in a given circuit. For 9 of the 12 applications, the compiler produces scaled ADPs that maximize at least one signal. For all applications, the compiler produces scaled ADPs in which at least half the signals occupy 50% of the operating range. This analysis demonstrates that the compiler can produce scaled ADPs with signals which exploit a significant fraction of the available operating ranges.

- **Amplitude of Values** (Section 10.5.5): I next investigate how effectively the scaled ADPs scale the fixed signals and values within the circuit. It is often desirable to scale fixed signals and values to have larger magnitudes since values with large magnitudes are more robust to noise and quantization error. It is generally not possible to maximize the magnitude of all fixed signals and values within a given circuit. For all applications, the compiler produces scaled ADPs that maximize at least one data field value. For all applications, the compiler produces scaled ADPs in which at least half the data field values are larger than `0.5` (the maximum possible magnitude is `1.0`). This analysis demonstrates that the compiler is, therefore, able to produce scaled ADPs with data field values and fixed signals which have relatively large magnitudes.

- **Scale Objective Function Value** (Section 10.5.6): The compiler is instructed to scale the ADP to minimize a `balanced` objective function which jointly maximizes quality and speed. I investigate the spread of `balanced` objective function values produced by the compiler. I find that the compiler can consistently find circuits that minimize the `balanced` objective function.

- **Scale Objective and Quality Measure Analysis** (Section 10.5.7): The compiler may identify multiple candidate scaling transforms which all minimize the `balanced` scaling objective function. I investigate the spread of the analog and digital quality measure values for each benchmark application – these quality measures together with the execution speed make up the `balanced` scaling objective function. This analysis reports multiple different quality measures and execution speeds because the compiler can minimize `balanced` scaling objective function in multiple different ways.

  In this analysis, I show that the compiler produces a wide range of scaling transforms that report various quality measures but all attain comparable scaling objective values. This analysis demonstrates that the compiler can identify multiple good candidate scaling transforms within the space of physically realizable, recoverable scaling transforms.

Viability of the Unscaled ADP (Section 10.6)

I investigate why the unscaled ADPs produced by the compiler cannot viably be executed on the HCDCv2. Sections 10.6.1, 10.6.2, and 10.6.3 investigate how severely the unscaled ADPs violate the frequency and operating range restrictions of the device.

For 11 benchmark applications, the unscaled ADPs contain values that exceed the maximum supported data field value – these applications cannot be written to the device at all. For all of the benchmark applications, the unscaled adps contain signals which violate the operating range and frequency restrictions of the device. These unscaled ADPs cannot be safely executed on the analog

device. All of these violations together concretely justify the need for a scaling transform.

Complexity of Scaling Transform (Section 10.7)

I then investigate the complexity of the scaling transform. I perform a detailed analysis of the characteristics of the scaled ADPs. The analysis summarizes the value ranges of the time and magnitude scale factors and identifies which proportion of the scale factors are unique. I find that the produced scaling transforms define between 4-30 unique (14-90 total) magnitude factors ranging from 0.01 to 3.46×10^4 and define time scale factors ranging from 0.09 to 0.63. This analysis demonstrates that the compiler derives complex scaling transforms with many distinct scale factors which span a wide range of values.

Compilation Outcomes and Result Quality (Section 10.8)

Section 10.5.6 reported that the scaled ADPs for a given benchmark report comparable `balanced` scale objective values. However, in practice, the produced waveforms vary significantly depending on the execution. This disparity between the static measure of circuit optimality and the actual end-to-end result quality may arise because the compilation procedure fails to consider the full range of hardware behaviors that may impact the end-to-end result.

This section investigates the relationship between the quality of the end-to-end result and the scaled ADP characteristics. The results of this analysis can be used to inform future compiler optimizations. For each investigated ADP characteristic, I investigate if the chosen calibration strategy affects the relationship between the studied ADP characteristic and the end-to-end result. In this analysis, I study the effect of the following ADP characteristics on end-to-end result quality:

- **Block Instance Selections**(Section 10.8.1): I investigate the relationship between the locations of the blocks in the scaled ADPs and the end-to-end result. For all of the benchmark applications, the subset of selected block instances has a sizable effect on the distribution of errors. This observation indicates that the location assignments derived in the place+route stage of compilation significantly affect result fidelity. In some cases, the `maximize_fit` calibration strategy can reduce, but not eliminate, these variances in the end-to-end result. For this reason, I believe it would be productive to engineer an extension to the place+route procedure that incorporates the effect of each block instance on the quality of the end-to-end result.

- **Quality Measures, Speed, and Scaling Objective** (Section 10.8.2): I investigate how the execution speed, scaling objective function value, and quality measures relate to the error of the end-to-end result. The primary goal of this analysis is to determine how predictive the scaling objective is of result fidelity. Another goal of this analysis is to understand whether any of the constitutive components of the scaling objective function (the speed and quality measures) strongly correlate with the end-to-end results. If any of these measures correlate strongly for all executions, then the `balanced` scaling objective likely can be simplified to a single term.

  In this analysis, I compute the Pearson correlation coefficient (PCC) between the `balanced` scaling objective value and the % normalized root-mean-squared error of the measured waveform. For 11 of the benchmarks, the `balanced` scaling objective function value correlates (PCC $\geq$ `0.5`) with the quality of the produced waveforms. For six of these benchmarks, the `balanced` scaling objective function value strongly correlates (PCC >`0.9`) with the quality of the end-to-end result. For 10 of 12 applications, the scaling objective function value correlates more strongly with the end-to-end result than the execution speed or any single quality measure. These observations indicate that the `balanced` scale objective is a good heuristic for identifying good scaling transforms. The `balanced` scale objective likely cannot be reduced to a single term and remain a good predictor.

  It's important to note that while the `balanced` scaling objective is a reasonably good predictor of quality, it is not a strong predictor of quality for all applications. The `balanced` scaling objective function is also less effective at predicting the quality of executions that use the

`maximize_fit` calibration strategy. I believe it would be productive to engineer objective functions which better predict result quality for these cases.

Potential of Alternate Scaling Objective Functions (Section 10.9)

I next explore the potential benefits of engineering an alternate scaling objective function. I introduce a new single-signal scaling objective function that maximizes a single signal or value, subject to a loose minimum AQM and DQM bound. I compare ADPs produced with this new scaling objective function with the ADPs produced with the `balanced` scaling objective function used by the compiler. If a subset of these single-signal maximizing executions outperforms the `balanced` scaling objective, then there is likely a signal or subset of signals which the compiler can maximize to deliver a good result. One potential future direction would be to develop a static analysis routine that identifies the subset of important signals in the ADP.

- **Performance** (Section 10.9.1): I compare the performance of this scaling objective function to the `balanced` scaling objective function used by the compiler. This analysis contrasts the quality, power, energy, and runtime characteristics of the two scaling objectives. This analysis aims to identify if these single-signal executions identify a new and desirable point in the tradeoff space.

  For five of the benchmarks, the single-signal objective function produces lower error executions. For four benchmarks, the single-signal objective function produces faster, lower energy, or lower power executions. These more performant executions produce waveforms with error characteristics that are better or competitive with the best-performing `balanced` executions discussed in Section 10.2. There are signals that, when maximized, unlock more performant or more accurate regions in the tradeoff space.

- **Signal Characteristics** (Section 10.9.2): I next investigate the characteristics of the signals maximized by the single-signal scaling objective function. This analysis aims to identify any patterns that can be leveraged to develop new scaling objective functions. Ten of the maximized signals implemented dynamical variables and eight of the signals implemented intermediate dynamical system expressions. Based on these observations, there doesn't seem to be a unifying heuristic that would identify these signals. I anticipate that a more thorough analysis would need to be developed to identify the signals to maximize.

Real-time Signals (Section 10.10)

I next compile and execute the two real-time signal processing applications introduced in Section 4.13 on the HCDCv2. Real-time signal processing applications are attractive computational targets for

benchmark	description	observation	time	diffeqs	funcs	nonlinear
cos	cosine	signal	20 su	2	1	no
cosc	dampened oscillator	oscillator amplitude	20 su	2	1	no
pend	pendulum	position of mass	20 su	2	1	yes
spring	two-mass spring system	position of mass 1	20 su	4	3	yes
vanderpol	vanderpol oscillator	signal	50 su	2	1	yes
heatN4X2	movement of heat	heat at point 2	120 su	4	1	no
forced	forced vanderpol oscillator	signal	10 su	4	1	yes
pid	PI controller	velocity	200 su	4	3	no
kalman	kalman filter	average	50 su	6	3	yes
gentog	genetic toggle switch	concentration of V	20 su	4	4	yes
smmrxn	michaelis menten reaction	complex	20 su	1	3	yes
bont4	bont	signal	20 su	5	1	no

Table 10.1: Dynamical system benchmarks used in evaluation.

analog devices such as the HCDCv2 since they have real-time performance requirements and typically operate in an energy-constrained environment.

Both evaluated signal processing applications accept an external analog signal as input, compute over the signal continuously in real-time, and emit a continually evolving result. I validate that the real-time signal processing applications perform computation on an externally provided real-time signal which exercises each application's functionality with acceptable accuracy.

10.1 Experimental Setup

Figure 10.1 presents an overview of the benchmark applications used in the evaluation. The benchmark applications contain between 2-6 differential equations and 1-4 straight-line functions. Seven of the twelve benchmarks implement non-linear dynamical systems. Each benchmark application produces one observable signal (`observation` column) and executes for 20-200 simulation time units (`time` column).

10.1.1 Compilation of Scaled ADPs

For each dynamical system benchmark, I configure the compiler to produce a total of 200 scaled ADPs. The 200 ADPs produced for each benchmark all implement the target dynamical system but may use different block instances, make different mode selections, target different calibration strategies, or specify different scaling transforms. I produced 200 ADPs for each benchmark by configuring the `LGraph` pass to generate ten ADPs and configuring the `LScale` pass to generate twenty scaled ADPs for each unscaled ADP. The `LScale` pass identifies ten distinct mode selections which minimize the `balanced` scale objective functions. Ten of the produced scaled ADPs target the `minimize_error` calibration strategy and ten of the produced scaled ADPs target the `maximize_fit` calibration strategy:

- **minerr**: ADPs compiled with the `minerr` configuration target analog blocks calibrated with

the `minimize_error` calibration strategy. This calibration strategy calibrates blocks so that their behavior adheres to the input-output relations specified in the ADS. The `minimize_error` strategy is the calibration strategy that is traditionally used for calibrating analog hardware.

- **maxfit**: ADPs compiled with the `maxfit` configuration target analog blocks calibrated with the `maximize_fit` calibration strategy. This calibration strategy allows for controlled behavioral deviations, provided the compiler can statically compensate for the behavioral deviations. The `maximize_fit` calibration strategy is co-designed to work together with the `LScale` pass of the compiler to produce accurate scaled ADPs.

The compiler produces scaled ADPs that minimize the `balanced` scale objective function. The `balanced` scale objective function jointly maximizes the quality and speed of the signals in the ADP. For all the scaled ADPs, the compiler compensates for the behaviors described in the devices' delta models and freely changes the mode selections for the blocks to better scale the circuit.

I limit the port and data field operating ranges to 95% of the entire operating range when scaling the circuit to limit the effect of difficult-to-characterize non-linearities on the computation. These non-linearities exist at the edges of the input space of most blocks.

10.1.2 Execution of Scaled ADPs

I evaluated the compiled benchmarks on the Sendyne development board, which interfaces with the HCDCv2 [132, 61, 51]. See Chapter 5 for more information on the computational blocks available on this platform.

Signal Acquisition and Analysis:

I collected waveforms for each benchmark using a Sigilent X1020E oscilloscope. I measure the amplitude of the analog waveform in mV and the time samples in wall-clock seconds. I apply the compiler-derived recovery transform to each signal to recover the original dynamical system dynamics from each measured waveform. I then compare the recovered signals to a reference simulation of the dynamical system computed by a standard digital differential equation solver that digitally simulates the dynamical system with high precision. As appropriate, I shift the measured signal in the time domain and apply minor changes to the time constant (scale by 0.98-1.02x) to account for otherwise uncharacterized deviations in hardware behavior.

Energy Consumption: I used an empirically-derived energy model provided by our collaborators to estimate the energy consumption of the device [50]. I use a model-based approach, as it is difficult to isolate the analog chip's power draw because it is embedded on a larger development board with other supporting circuitry. The maximum power consumption of the device is 1.2 mW.

10.1.3 Overview of Statistical Measures

The analyses presented in this chapter use a variety of statistical measures to summarize distributions of outcomes. The chosen statistical measures are robust to outlier results and adequately capture both the typical outcome and the spread of outcomes:

- **Median**: The median value is the value at the 50% percentile of the dataset. Half of the values in the dataset are greater than the median value, and half of the values in the dataset are smaller than the median value.

- **First (Q1) and Third (Q3) Quartiles**: The first and third quartiles are values at the 25% and 75% percentile of the dataset. The Q1 and Q3 quartiles quantify the lower and upper values of the middle 50% of values in the dataset. These quartiles, therefore, exclude any extreme values present in the dataset.

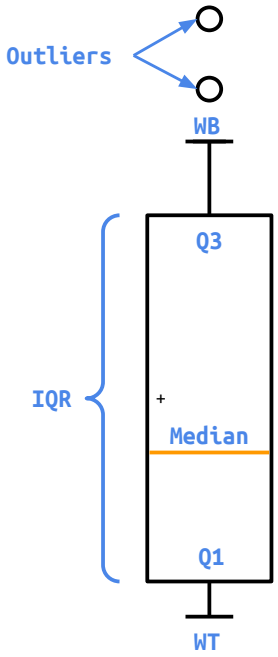
- **Interquartile Range (IQR)**: The interquartile range (IQR) quantifies the spread of values. The IQRis the difference between the third and first quartile of the dataset (`Q3-Q1`). The IQRquantifies the typical spread of outcomes in the dataset.

- **Low (WT) and And High (WB) Values**: The low and high values of the dataset quantify the lowest and highest non-outlier values. All values outside of the range `[median - 1.5*IQR,median + 1.5*iqr]` are reported as outliers.

- **Minimum and Maximum Values**: The minimum and maximum values of the dataset quantify the lowest and highest values (including outliers). These metrics are the simple minimum and maximum of the dataset.

  For some results, only the minimum and maximum values are reported. These results are written as a range `max(x) - min(x)` if the minimum and maximum values differ. If the minimum and maximum values are the same, then only one number is reported.

Box Plot Visualizations



The above statistics are typically presented as a table of values or a box and whisker plot (left figure). The figure to the left is an annotated version of the box and whisker plot. The orange line on the plot indicates the median value of the distribution. The top and the bottom of the box are the third and first quartiles of the data, and the top and bottom notches are the high and low values of the data. Any points above or below the box and whisker plot are outlier values. Typically, the outliers are excluded from box and whisker plots as they reduce the readability of the plot.

Statistical Measures and End-to-End Quality

The presented analyses often discuss the distribution of % root-mean-squared (% rmse) errors for all executions of a particular type. These analyses use the following terms to describe the distribution of observed errors:

- *Best-performing execution* (`minimum`) – The best-performing execution is the execution with the lowest % rmse. The best-performing execution is the most accurate execution produced by the compiler. This execution is the best the compiler can do for the target benchmark for a given execution type.
- *Typical execution* (`median`) – This analysis reports the median error for all `minerr` and all `ideal` executions. The execution with the median error can be thought of as the error an end user might see from a typical execution. The typical error is the % rmse associated with the typical execution.

benchmark	execution	%rmse	runtime (ms)	power (mW)	energy (uJ)
cos	maxfit	0.05	0.52	0.20	0.10
cosc	minerr	1.96	0.25	0.40	0.10
pend	minerr	0.15	0.50	0.45	0.23
spring	minerr	0.99	1.09	1.03	1.13
vanderpol	minerr	1.19	1.25	0.74	0.92
forced	maxfit	0.69	0.25	0.84	0.21
heatN4X2	minerr	0.01	1.50	0.76	1.14
pid	maxfit	0.63	5.95	0.86	5.09
kalman	maxfit	0.04	2.50	0.86	2.16
smmrxn	maxfit	4.50e-04	0.50	0.53	0.27
gentog	maxfit	1.72	1.82	1.10	2.00
bont4	maxfit	3.46e-06	0.26	0.93	0.24

Table 10.2: Quality, runtime, power, and energy measures for best performing ADPs.

- *Error spread* (`iqr`) – This analysis reports the IQR for all `minerr` and `ideal` executions. The IQR captures the spread of errors for a particular execution type (excluding any outliers). Execution types which produce executions with a small error IQRare able to more consistently deliver executions with similar error characteristics.

10.2 Quality, Runtime, Power, and Energy

Table 10.2 presents the performance characteristics of the best-performing (lowest % rmse) dynamical system simulations when executed on the HCDCv2 board. Column 1 reports the type of execution (`minerr` or `maxfit`) which has the lowest error. Columns 2-5 report the normalized root mean-squared error (`% rmse`), runtime, power, and energy consumption of each benchmark.

Quality: The normalized root-mean-squared error ranges from $3.46 \cdot 10^{-6}$% to 1.96% of the dynamic range of the signal for the best performing execution depending on the benchmark. This high degree of agreement between the reference and measured signal indicates that the compiler is capable of producing highly accurate ADPs for a wide range of benchmark applications. The low error also confirms that the computed mapping between wall-clock time and simulation time is within the 2% error margin described in the experimental setup. This indicates that the statically derived runtime estimation is an accurate estimate of the time required to execute the benchmark applications.

None of the applications report a zero % rmse – this is not unexpected as the compilation procedure does not capture and compensate for all possible behaviors present in the analog device. For example, the compiler does not model or compensate for the frequency-gain characteristics for each block instance. The compiler captures (with delta models) but does not compensate for any residual signal biases present in the blocks post-calibration. The compiler also does not eliminate the physical phenomena captured in the hardware specification. For example, the noise present in the block may still affect the computation as the scaling procedure only reduces the effect of the

noise on the computation.

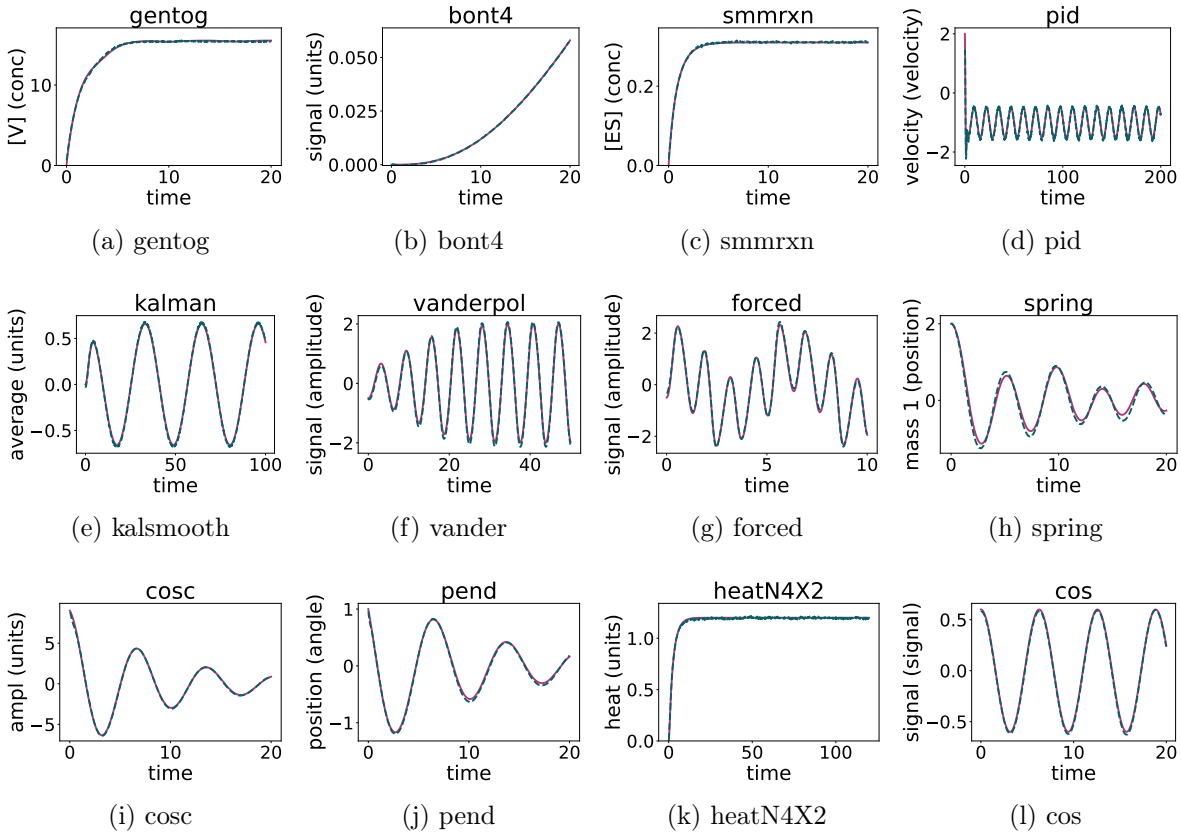
Runtime: The execution times range from 0.25 to 5.95 milliseconds for the best-performing executions. The execution times vary significantly across benchmarks because different benchmarks are executed for different numbers of simulation time units. Variations in the time scale factor value also affect the execution time of the benchmark.

The time required to run each benchmark does not grow with the complexity of the benchmark. For example, the `bont` benchmark contains five differential equations and one function but completes in less wall-clock time than the much simpler `cos` benchmark (both benchmarks execute for 20 simulation time units). The HCDCv2 is also capable of executing non-linear systems in less time than linear systems. For example, the nonlinear `smmrxn` benchmark executes to completion in less time than the linear `cos` benchmark (both benchmark execute for 20 su). In contrast, traditional digital simulation techniques take longer to simulate larger dynamical systems and experience performance degradations when simulating non-linear systems.

Power Consumption: The power consumption ranges from 0.20 to 1.10 milliwatts depending on the benchmark for the best performing executions. There are variations in power consumption because only the enabled components draw power. The power consumption of a block may also vary depending on the block mode and the integration speed of the scaled circuit.

Energy Consumption: The total energy consumption for each execution is equal to the power consumption multiplied by the runtime. The energy consumption ranges from 0.10-5.09 μJ depending on the benchmark for the best performing executions. Overall, these benchmark applications consume very little energy.

Best-Performing Execution: For the `cos`, `pid`, `kalman`, `forced`, `smmrxn`, `gentog`, and `bont4` benchmarks, the `maxfit` calibration strategy produces the best-performing execution. For the `cosc`, `pend`, `spring`, `vanderpol`, and `heatN4X2` benchmarks, the `minerr` calibration strategy produces the best-performing execution. Note that sometimes the `minerr` produces better executions than `maxfit` executions. Refer to Section 10.3.4 for a deeper analysis of the relationship between the calibration strategy and the end-to-end result.



(a) gentog (b) bont4 (c) smmrxn (d) pid

(e) kalsmooth (f) vander (g) forced (h) spring

(i) cosc (j) pend (k) heatN4X2 (l) cos

Figure 10-1: Measured and reference waveforms for the lowest-error ADPs. Each reported waveform is from a `minerr` or `maxfit` execution. red lines are reference signals. green lines are measured signals. Note that measured signals are largely visually indistinguishable from reference signals.

10.2.1 End-to-End Result Quality (% rmse)

Figure 10-1 presents a qualitative comparison of the analog signals measured from the HCDCv2 (green) with the reference simulation (red) for the best performing ADPs from the previous section. I obtain the reference simulations by solving the benchmark applications with a high precision digital differential equation simulator. In all cases, the analog signal closely tracks the reference simulation, as can be seen by the fact that the signals are essentially indistinguishable. In Figure 10-1, only a single line is visible for each benchmark application – this is because the reference waveforms and measured waveforms are superimposed on top of one another. The most visible deviations occur for the `spring` benchmark. Small deviations in the measured signal are acceptable in the above benchmarks as the benchmark applications themselves allow for some error:

Physical Systems: Dynamical systems model physical phenomena (e.g., physics and biology simulations) that are inherently approximate: the constants are often derived from empirical measurements and, in many cases, the dynamics are approximations of physical phenomena. With these

systems, state variable trajectories are typically inspected visually.

Control Systems: Control systems are typically designed with some high-level objective in mind. For the `pid` program, the objective is to attenuate any perturbations. For the `kalman` program, the goal is to track an input signal. Both the analog and reference implementations of these computations meet these objectives.

10.3 Compiler Optimizations and Result Quality

I next investigate the effect of various compiler optimization and compiler design decisions on the end-to-end result quality.

What if the compiler doesn't scale the circuit? Section 10.3.1 investigates the impact of disabling the circuit scaling compilation pass on the end-to-end result quality. With circuit scaling disabled, none of the benchmarks could be executed on the hardware. The circuit scaling procedure is therefore integral to mapping computations onto the analog hardware.

What if the compiler scales the circuit but doesn't aggressively change the mode?: Section 10.3.2 investigates of the effect of disabling master expression support in the `LScale` pass. This modification to compilation bars the scaling procedure from selecting any modes which are not already specified in the unscaled ADP. In this analysis, the delta model compensation optimization is also disabled.

With aggressive mode selection disabled, only the `bont4`, `heatN4X2`, `cosc`, and `smmrxn` benchmarks produce results which agree with the reference signal. The `cos` and `forced` benchmarks could not be compiled at all. The compiler's ability to aggressively change the block modes significantly improves its ability to scale the circuit effectively.

What if the compiler doesn't compensate for behavioral variations?: Section 10.3.3 investigates the effect of delta model compensation on the end-to-end result. With delta model compensation disabled, only the `vanderpol` benchmark produces an execution that closely matches the reference executions. With delta model compensation disabled, ten of the twelve benchmarks report a higher median error, and five of the ten benchmarks report a higher spread of errors. Therefore, `LScale`s ability to compensate for behavioral variations produces lower-error executions on average and can produce executions with an overall lower error.

What if the compiler targets a calibration strategy that has been co-designed to work with the circuit scaling procedure?: Section 10.3.4 also investigates the effect of the calibration strategy on the end-to-end result. This analysis compares executions which target the `minimize_error` calibration strategy with executions which target the `maximize_fit` calibration strategy. The `minimize_error` calibration strategy is a classical calibration strategy that calibrates blocks to implement the input-output relations described in the ADS. In contrast, the `maximize_fit`

calibration strategy is specially designed to work with the scaling procedure. It allows blocks to deviate from the desired input-output relation provided the `LScale` compilation pass can correct the behavioral variations with delta model compensation techniques.

With the co-designed `maximize_fit` calibration strategy, eight of the twelve benchmarks report a lower median error, and five of the twelve benchmarks report a lower spread of errors (four are comparable). For seven of the ten benchmarks, the lowest-error execution is a `maximize_fit` execution. Therefore, the co-designed calibration strategy more consistently produces comparable or lower error results for a significant fraction of applications.

10.3.1 Effect of Scaling Transform

What if the compiler doesn't scale the circuit?

To explore the effect of circuit scaling on the quality of the end-to-end execution result, I execute the circuit scaling pass with an identity transform where all signals and values are scaled by one. I introduce a new type of execution to perform this analysis:

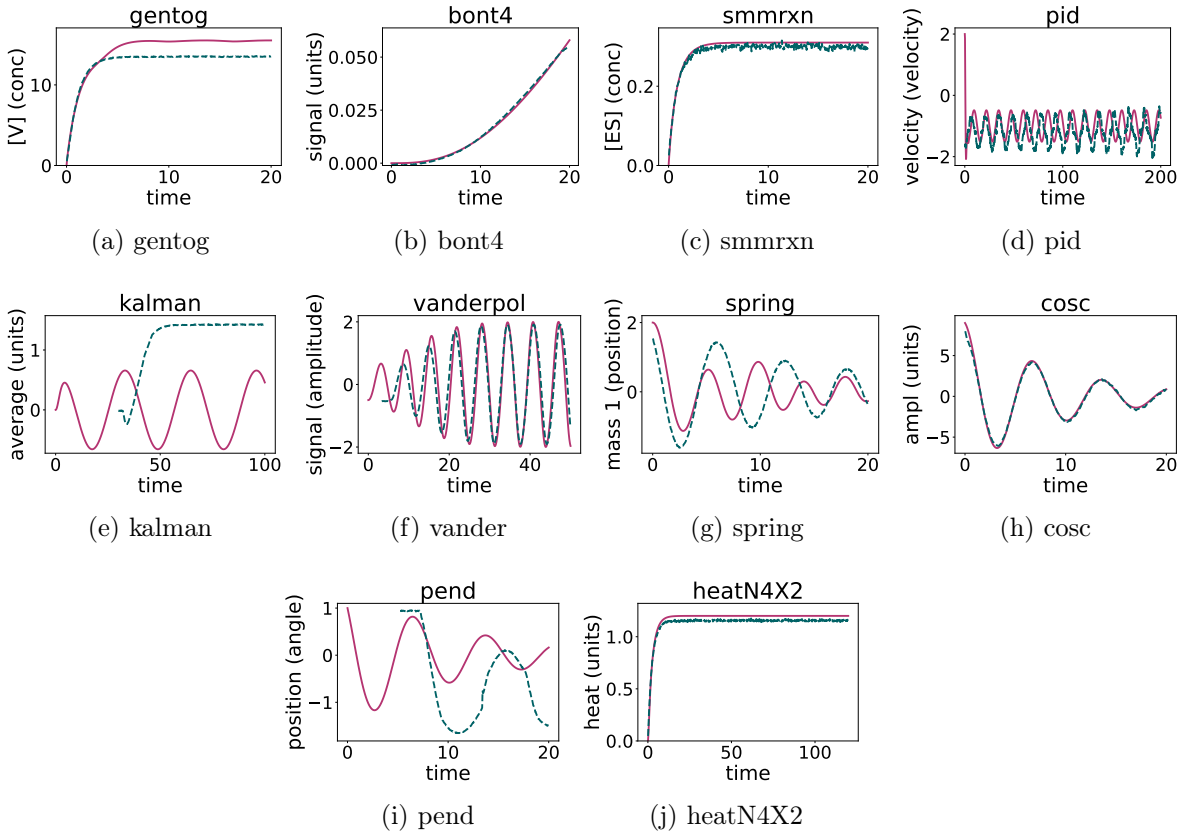
- **noscale**: The `noscale` executions isolate the effect of the scaling transform on the fidelity of the result. To produce these executions, I direct the circuit scaling pass to set each time and magnitude scale factor to one. I also disable delta model compensation since the compensation process requires a non-unitary scaling transform.

The compiler is unable to generate any ADPs that respect the physical constraints of the device with circuit scaling disabled. All benchmarks assign one or more data fields to values outside the supported data field value ranges or violate the device's operating range and frequency constraints. These results suggest that the circuit scaling pass enables the compiler to successfully map dynamical systems to the analog hardware.

10.3.2 Effect of Mode Selection

What if the compiler scales the circuit, but doesn't aggressively change the mode?

I next investigate what effect the inclusion of master expressions has on the end-to-end result. The master expression elicitation procedure enables the circuit scaling pass to more flexibly change the block modes in the ADP and to account for behavioral deviations in the device. This analysis aims to identify the effect the mode selection feature has on the quality of the end-to-end result. This analysis introduces a new type of execution that scales a given ADP without using master expressions.



(a) gentog (b) bont4 (c) smmrxn (d) pid

(e) kalman (f) vander (g) spring (h) cosc

(i) pend (j) heatN4X2

Figure 10-2: Benchmark executions on HDACv2 analog board. red lines are reference signals. green lines are measured signals.

- **nomaster**: For these executions, the compiler scales the circuit without using master expressions. Without master expressions, the compiler cannot aggressively change the mode or perform delta model compensation when producing `nomaster` ADPs. The compiler may only use the block modes listed in the unscaled ADP when scaling the circuit. I execute each produced scaled ADP on the device after the device has been calibrated with the `minimize_error` calibration strategy. The `minimize_error` calibration strategy calibrates the blocks to implement the input-output relations defined in the ADS. Blocks calibrated with this strategy require less delta model compensation.

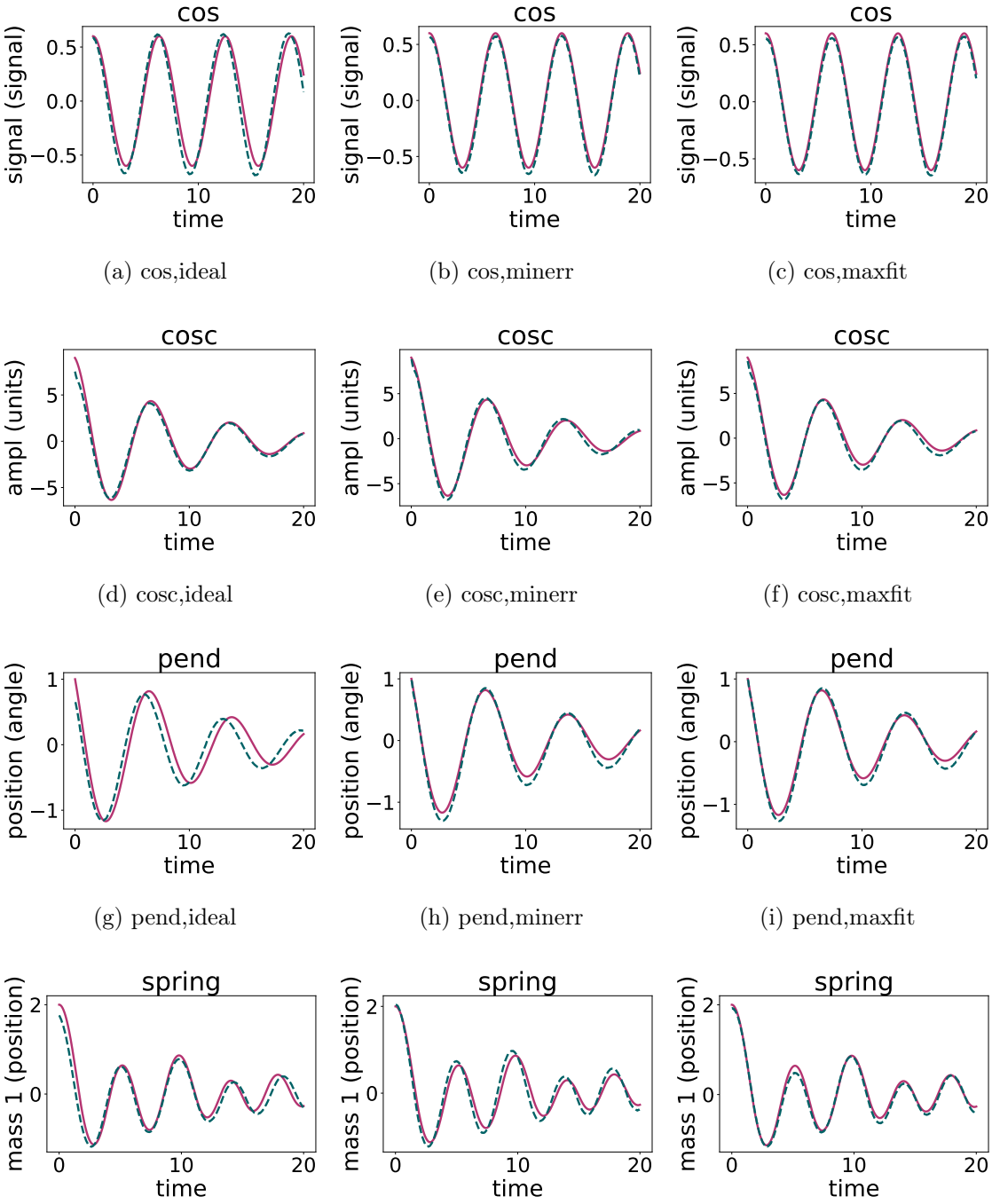
Figure 10-2 presents a comparison of the best-performing (lowest % rmse) waveform (green) to the reference signal (red) for the `nomaster` executions. The `LScale` pass was unable to scale `forced` and `cos` benchmarks without using a master expression. The measured signals for the `bont4`, `heat`, `smmrxn`, and `cosc` benchmarks track the reference signals fairly closely. All other benchmarks deviate significantly from the expected dynamics. These waveform deviations likely occur because the compiler cannot freely change block mode during the circuit scaling process. Recall

		LScale pass runtime (s)				best-performing execution			
program	execution	median	iqr	min	max	%rmse	runtime (ms)	power (mW)	energy (uJ)
bont4	minerr/maxfit	6.87	6.23	0.19	16.25	3.461e-06	0.26	0.93	0.24
bont4	nomaster	6.87	6.23	0.19	16.25	1.319e-04	0.25	0.82	0.21
heatN4X2	minerr/maxfit	3.10	1.78	0.22	8.83	1.193e-02	1.50	0.76	1.14
heatN4X2	nomaster	3.10	1.78	0.22	8.83	3.956e-01	1.50	0.74	1.11
smmrxn	minerr/maxfit	1.69	1.46	0.09	3.71	4.495e-04	0.50	0.53	0.27
smmrxn	nomaster	1.69	1.46	0.09	3.71	2.063e-02	0.50	0.53	0.26
cosc	minerr/maxfit	1.45	1.16	0.08	3.6	1.958e+00	0.25	0.40	0.10
cosc	nomaster	1.45	1.16	0.08	3.6	1.693e+01	0.25	0.38	0.10

Table 10.3: `LScale` compilation times and quality, runtime, power and energy measurements for accurate `nomaster` executions

that the HCDCv2 blocks often offer high dynamic range modes that both change the block's input-output relation and significantly expand the port operating ranges. Without master expressions, the compiler cannot select these modes unless they are already selected in the unscaled ADP. Therefore, the inclusion of master expressions in the scaling procedure is crucial for attaining good results for a significant subset of benchmark applications.

For the benchmark applications which report `nomaster` executions that execute accurately, I observe negligible `LScale` pass compilation time improvements and no appreciable execution time or power improvements. Table 10.3 presents the performance characteristics for the accurate `nomaster` executions. Including master expressions in the scaling procedure incurs no additional overhead. The best-performing `nomaster` executions report higher error than the best performing `minerr` and `maxfit` executions for the above benchmarks. The `nomaster` executions report slightly lower power and energy consumption figures because the higher energy block modes are likely not selected by the `LScale` pass.



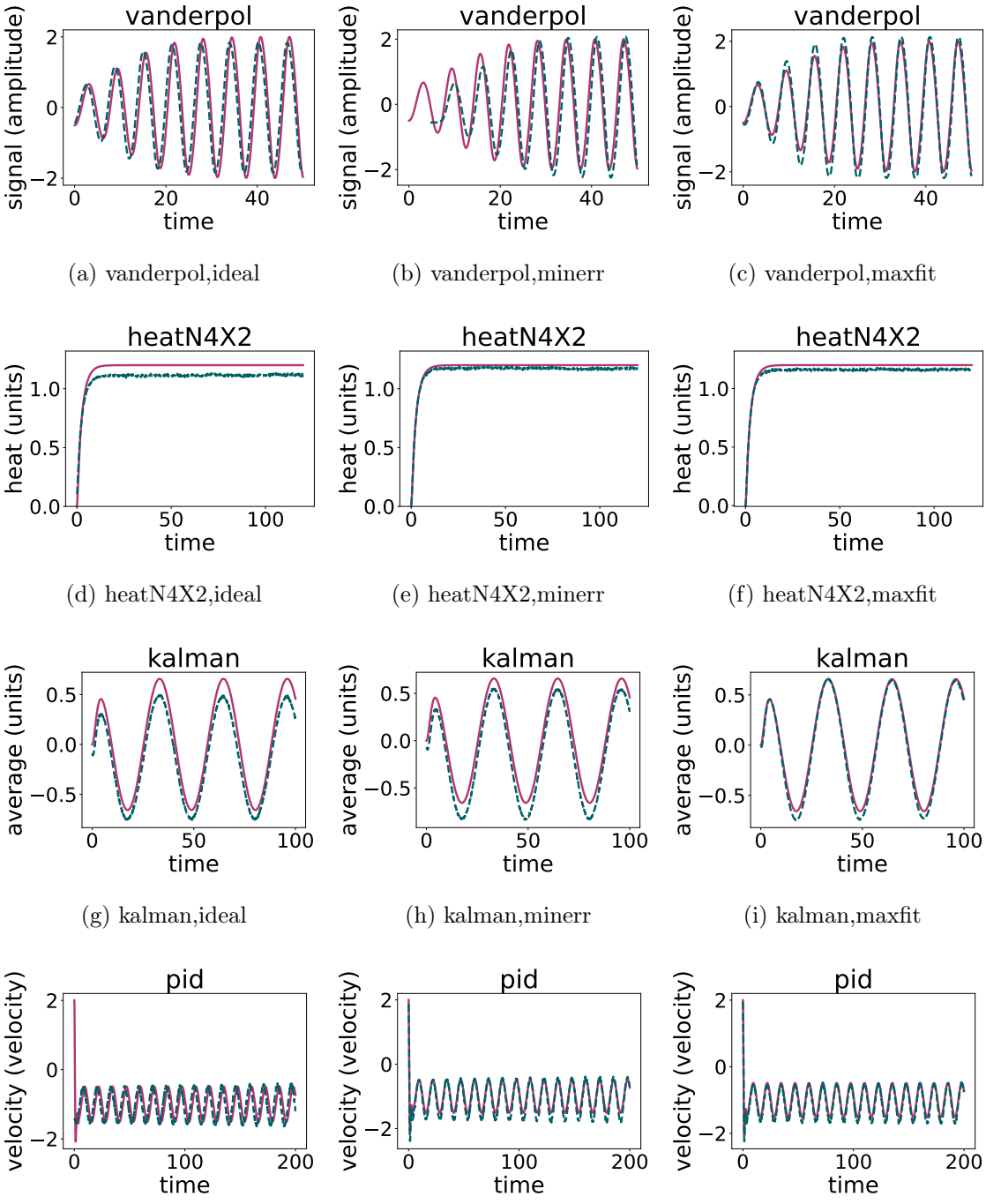
(a) cos,ideal (b) cos,minerr (c) cos,maxfit

(d) cosc,ideal (e) cosc,minerr (f) cosc,maxfit

(g) pend,ideal (h) pend,minerr (i) pend,maxfit

(j) spring,ideal (k) spring,minerr (l) spring,maxfit

Figure 10-3: Waveforms for `ideal`, `minerr`, and `maxfit` executions of the `cos`, `cosc`, `pend`, and `spring` benchmarks. Each above execution is the median %rmse execution. red lines are reference signals. green lines are measured signals.



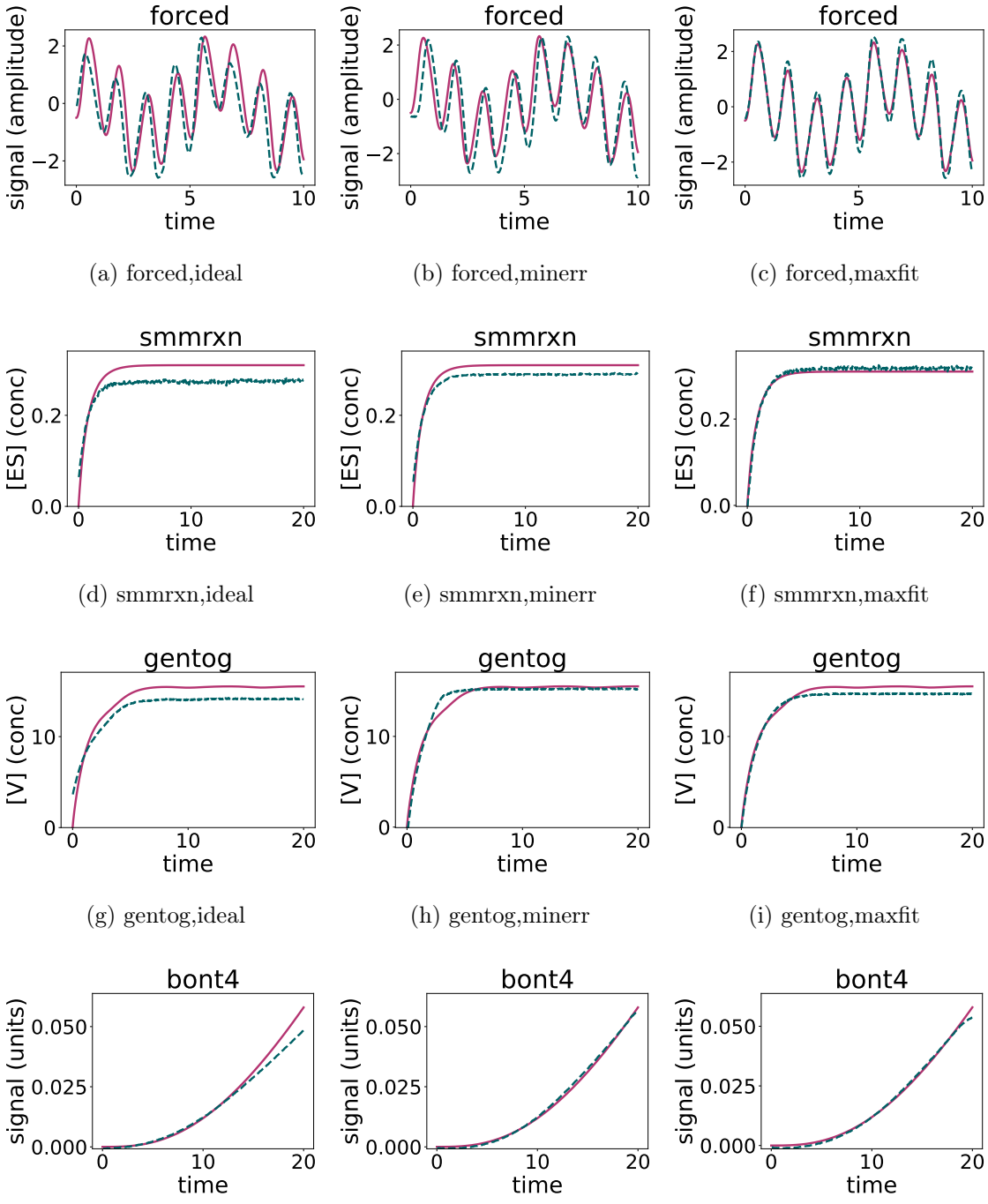
(a) vanderpol,ideal (b) vanderpol,minerr (c) vanderpol,maxfit

(d) heatN4X2,ideal (e) heatN4X2,minerr (f) heatN4X2,maxfit

(g) kalman,ideal (h) kalman,minerr (i) kalman,maxfit

(j) pid,ideal (k) pid,minerr (l) pid,maxfit

Figure 10-4: Waveforms for `ideal`, `minerr`, and `maxfit` executions of the `vanderpol`, `heatN4X2`, `kalman`, and `pid` benchmarks. Each above execution is the median %rmse execution. red lines are reference signals. green lines are measured signals.



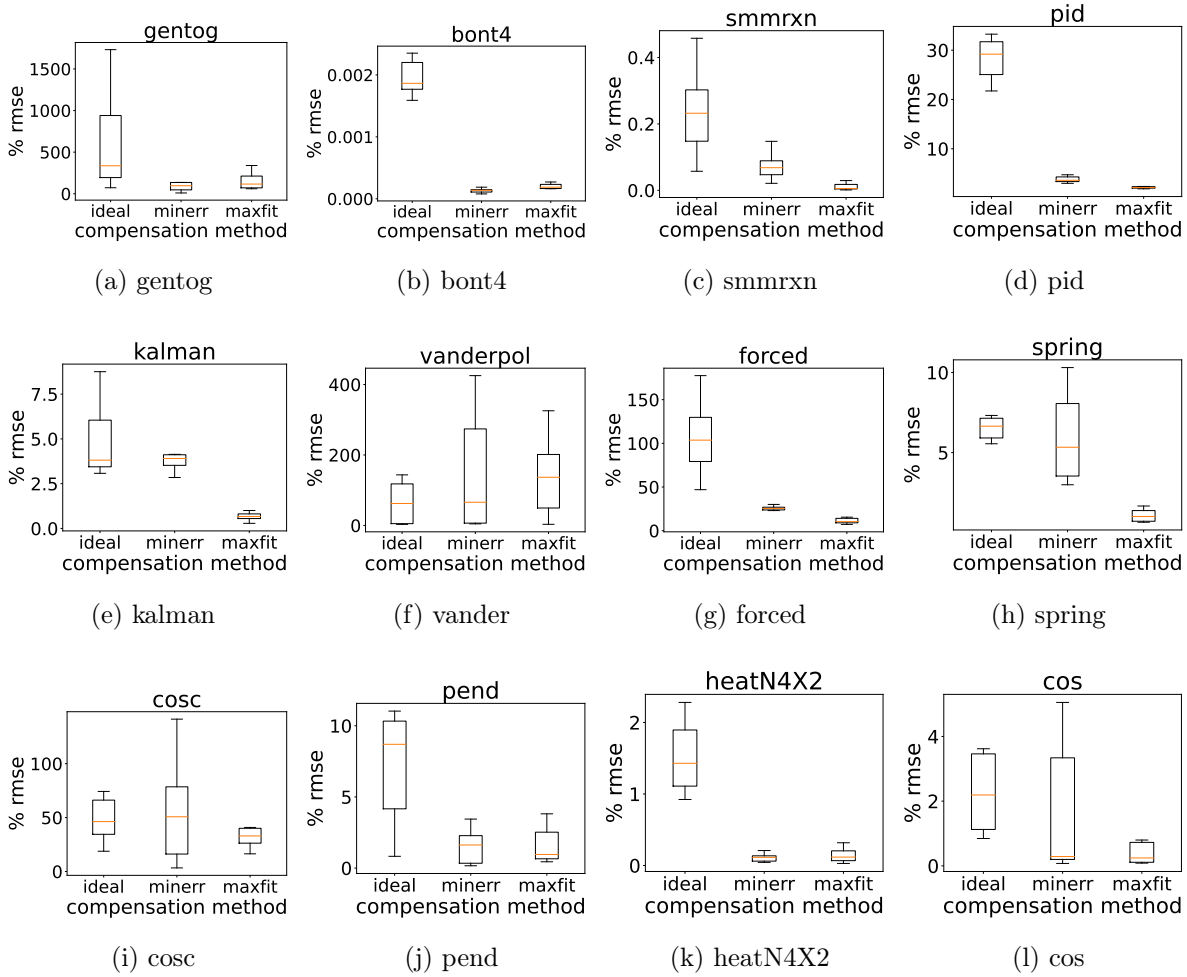
(a) forced,ideal (b) forced,minerr (c) forced,maxfit

(d) smmrxn,ideal (e) smmrxn,minerr (f) smmrxn,maxfit

(g) gentog,ideal (h) gentog,minerr (i) gentog,maxfit

(j) bont4,ideal (k) bont4,minerr (l) bont4,maxfit

Figure 10-5: Waveforms for `ideal`, `minerr`, and `maxfit` executions of the `forced`, `smmrxn`, `gentog`, and `bont4` benchmarks. Each above execution is the median %rmse execution. red lines are reference signals. green lines are measured signals.



(a) gentog (b) bont4 (c) smmrxn (d) pid

(e) kalman (f) vander (g) forced (h) spring

(i) cosc (j) pend (k) heatN4X2 (l) cos

Figure 10-6: Distribution of % rmses for `ideal`, `minerr`, and `maxfit` executions. The y-axis reports the % rmse and the x-axis reports the type of execution. Refer to Section 10.1.3 for an overview of box plots.

10.3.3 Effect of Delta Model Compensation

What if the compiler scales the circuit and uses master expressions, but does not correct for behavioral deviations?

I next investigate the effect of the delta model compensation feature from the circuit scaling pass. Delta model compensation enables the circuit scaling pass to consider the empirically observed behavioral deviations present in the device when scaling the circuit. To evaluate this compiler feature, I introduce a new kind of execution:

- `ideal`: For these executions, the compiler scales the circuit without performing delta model compensation. Instead, the circuit scaling pass assumes each block perfectly implements the idealized input-output relation defined by the `rel` statements in the ADS block specification. Each scaled `ideal` ADP targets the `minimize_error` calibration strategy. Blocks calibrated with this strategy require less delta model compensation.

This analysis compares the `ideal` executions described above with the `minerr` executions introduced in Section 10.1. The `minerr` executions also target the `minimize_error` calibration strategy but are produced with delta model compensation enabled. This analysis uses the error metrics described in Section 10.1.3 to describe the distribution of errors for each execution.

Best-Performing Executions

The `ideal` and `minerr` plots in Figures 10-3, 10-4, and 10-5 report the best-performing executions for the `ideal` and `minerr` execution types respectively. For all applications except `vanderpol`, the best-performing `ideal` execution does not track the signal as closely as the `minerr` execution. For the `cosc`, `pend`, `spring`, and `pid` executions, the initial condition is incorrect but the overall trajectory tracks well. For the remaining executions, the overall trajectory for the `ideal` execution does not track the reference signal as well as the `minerr` execution. This indicates that delta model compensation enables the compiler to identify scaled ADPs which produce lower end-to-end errors than the `ideal` executions.

Typical Executions

The `ideal` and `minerr` boxplots in Figure 10-6 present the distribution of errors for the `ideal` and `minerr` executions respectively. The error for the typical execution is indicated with the orange line on each box plot. For most of the benchmarks, the delta model compensation optimization improves the overall fidelity of the result. For all applications except the `vander` and `cosc` benchmarks, the `minerr` executions report a lower typical execution error than the `ideal` executions. This observation indicates that delta model compensation enables the compiler to produce lower-error

results. I attribute this to the fact that the compensation procedure can correct for instance- and mode-dependent behavioral variations. For the `vander` and `cosc` benchmarks, the `minerr` and `ideal` executions report comparable typical execution error. For these applications, delta model compensation doesn't significantly improve the average-case error.

Consistency of Executions

The `ideal` and `minerr` boxplots in Figure 10-6 present the distribution of errors for the `ideal` and `minerr` executions respectively. The consistency of an execution is indicated by the size of the IQR – a larger IQR indicates the executions do not consistently attain the same error. For many of the benchmarks, delta model compensation reduces the spread of errors observed across executions. For the `gentog`, `bont4`, `smmrxn`, `pid`, `kalman`, `forced`, `pend`, and `heatN4X2` executions, the error spread is lower for the `minerr` executions than for the `ideal` executions.

For the `vander`, `spring`, `cosc`, and `cos` benchmarks, the `minerr` executions report a larger spread of errors than the `ideal` executions. One possible explanation is that delta model compensation reduces the set of viable block modes for a subset of the block instances because it introduces non-idealities that cannot be compensated for by the compiler. For example, the delta model compensation procedure might introduce non-unity terms into the derivatives of the `integ` blocks. The compiler cannot correct these terms if the ADP does not introduce blocks that scale the derivative signals.

10.3.4 Effect of Calibration Strategy

> *What if the compiler targets a calibration strategy that has been co-designed to work with the circuit scaling procedure?*

I next investigate the effect of calibration strategy on the overall accuracy of the computation. This analysis investigates if this co-designed (`maximize_fit`) calibration strategy produces lower error executions than the classical (`minimize_error`) calibration strategy:

- **`maximize_fit` strategy** – The `maximize_fit` calibration strategy is a new calibration strategy which was designed to work in concert with the `LScale` compilation pass. The `maximize_fit` strategy prioritizes eliminating point errors over implementing the ADS input-output relations. Blocks calibrated with this strategy can produce scaled output signals and scale parts of the implemented input-output relations. Note that one downside of the `maximize_fit` strategy is that it may waste parts of the input and output port operating range if the behavior deviates significantly.

- `minimize_error` **strategy** – The `minimize_error` calibration strategy is a traditional calibration strategy calibrates the device to implement the input-output relations from the ADS. The `minimize_error` strategy effectively uses the entire operating range of each block since it does not deviate from the provided specification, but it may fail to eliminate point errors or errors in parts of the input space.

This analysis compares the `maxfit` and `minerr` executions introduced in Section 10.1. This analysis uses the error metrics described in Section 10.1.3 to describe the distribution of errors for each execution.

Best-Performing Executions

The `minerr` and `maxfit` plots in Figure 10-3, Figure 10-4, and Figure 10-5 present the best-performing execution for the `minerr` and `maxfit` executions. For the `pend`, `spring`, `vanderpol`, `kalman`, `pid`, `forced`, and `smmrxn` executions, the `minerr` executions do not track the reference signal as closely as the `maxfit` executions. One possible explanation is that the `minimize_error` calibration strategy introduces small errors into the input-output relation which are attenuated away when the block is calibrated with the `maximize_fit` calibration strategy.

For the `cos`, `cosc`, `heatN4X2`, and `gentog` benchmark, both the `minerr` and `maxfit` executions track the reference signal equally well. For these executions, the compiler was able to find a scaled ADP for both calibration strategies which delivers a good result.

For the `heatN4X2` benchmark, the `minerr` execution produces a better result than the `maxfit` execution. This situation may occur when the compiler happens to identify a combination of block instances and block modes which can be calibrated effectively with the `minimize_error` calibration strategy. Because the `minimize_error` strategy maximizes the usable operating range of each port, it's likely to produce a good result provided the calibration algorithm can eliminate all of the block errors. For this application, the compiler happens to identify a scaled ADP which has this property.

Typical Executions

The `minerr` and `maxfit` boxplots in Figure 10-6 present the distribution of errors for the `minerr` and `maxfit` executions respectively. The error for the typical execution is indicated with the orange line on each box plot. For the `smmrxn`, `pid`, `kalman`, `forced`, `spring`, `cosc`, and `pend` benchmarks, the `maxfit` executions report a lower median error than the `minerr` executions. For the `gentog`, `heatN4X2`, and `cos` benchmarks, the `maxfit` executions report a comparable median error to the `minerr` executions. Therefore, for 10 of the 12 benchmarks, the `maxfit` executions produce better or similar results as the `minerr` executions. This indicates that the co-designed calibration strategy is able to to produce a lower-error results than the classical calibration strategy. The `maxfit` executions

may enjoy a better result on average for many applications because the maximize_fit calibration strategy produces more blocks which are well-behaved on-average.

The vander and bont4 benchmarks, the maxfit executions report a higher median error than the minerr executions. Note that for the bont4 benchmark, the median error is only slightly higher for the maxfit executions than the minerr executions. One possible explanation is that, for these benchmarks, that wasting parts of the operating range has a larger effect on the end-to-end result than point errors for a higher proportion of ADPs.

Consistency of Executions

The minerr and maxfit boxplots in Figure 10-6 present the distribution of errors for the minerr and maxfit executions respectively. The consistency of an execution is indicated by the size of the IQR – a larger IQR indicates the executions do not consistently attain the same error. For the smmrxn, pid, kalman, vander, spring, txcosc, and cos benchmarks, the maxfit executions report a smaller spread of errors than the minerr executions. The maximize_fit calibration strategy is likely able to more consistently deliver the same results across different block modes and block instances because it adopts a relaxed functional specification for each block which allows for controlled deviations. This likely enables the maximize_fit calibration strategy to identify a low error configuration for a larger fraction of blocks.

For the bont4, pend, and forced benchmarks, the maxfit executions report a similar spread of errors as the minerr executions. Note that the maxfit executions for the pend and forced benchmarks the achieve a lower median error than the minerr executions. Therefore, for nine of the twelve benchmarks, the maxfit executions either more consistently deliver the same error or deliver a lower error at comparable consistency when compared to the minerr executions.

For the heatN4X2 and gentog benchmarks, the maxfit executions report a larger spread of errors than the minerr executions. Note that in both these applications, the increase in the error spread is relatively minor. One possible explanation is that for these benchmarks, the large range of behavioral variations found in the maximize_fit executions has a significant impact on the quality of the produced scaling transforms. The maxfit executions may therefore produce a larger error spread as the functions implemented by the blocks calibrated with the maximize_fit calibration strategy may vary significantly across block instances and block modes.

	LGraph runtimes (s)				LScale runtimes(s)			
benchmark	median	iqr	min	max	median	iqr	min	max
cos	0.31	0.01	0.30	1.49	0.13	0.16	0.03	0.28
cosc	0.12	0.03	0.10	4.59	0.76	0.37	0.41	0.98
pend	0.14	1.19	0.11	5.72	0.94	0.33	0.50	1.31
spring	0.37	0.01	0.36	23.45	5.74	1.13	4.17	7.51
vanderpol	0.30	0.03	0.29	13.96	1.91	0.40	1.38	2.73
heatN4X2	0.57	0.02	0.55	31.84	1.92	1.04	1.13	3.03
pid	0.33	0.01	0.32	18.63	3.63	1.00	2.54	5.47
kalman	0.35	0.02	0.34	18.72	3.56	1.13	2.49	4.43
forced	0.40	0.06	0.39	22.98	3.62	0.63	2.42	4.61
smmrxn	0.18	0.03	0.15	8.03	0.82	0.47	0.49	1.22
gentog	0.36	0.03	0.35	24.72	6.13	2.15	4.02	8.58
bont4	0.38	0.03	0.36	22.74	3.63	0.73	2.79	5.07

Table 10.4: Compilation times for the LGraph and LScale compilation passes.

	synthesis	assembly				place+route			
benchmark	time	median	iqr	min	max	median	iqr	min	max
cos	0.21	0.06	0.00	0.06	0.06	0.31	0.01	0.30	1.06
cosc	0.50	0.07	0.00	0.07	0.07	0.12	0.00	0.11	3.81
pend	0.57	0.06	0.00	0.06	0.06	0.14	1.19	0.11	4.94
spring	1.34	0.07	0.00	0.07	0.07	0.37	0.01	0.36	21.85
vanderpol	1.48	0.07	0.00	0.07	0.07	0.30	0.03	0.29	12.25
heatN4X2	1.98	0.08	0.00	0.08	0.08	0.57	0.02	0.55	29.60
pid	1.84	0.08	0.00	0.08	0.08	0.33	0.01	0.32	16.54
kalman	1.42	0.07	0.00	0.07	0.07	0.35	0.02	0.34	17.04
forced	2.25	0.07	0.00	0.07	0.07	0.40	0.06	0.39	20.50
smmrxn	1.30	0.07	0.00	0.07	0.07	0.18	0.03	0.15	6.50
gentog	1.84	0.08	0.00	0.08	0.08	0.36	0.03	0.35	22.60
bont4	1.16	0.08	0.00	0.08	0.08	0.38	0.03	0.36	21.34

Table 10.5: LGraph performance breakdown by compilation pass.

10.4 Compilation Time

Table 10.4 presents the compilation times for the LGraph and LScale compilation passes. Columns 2,3,4, and 5 report the median time, the execution time spread (inter-quartile range), and the minimum and maximum times required for LScale to synthesize unscaled circuits. Columns 6,7,8, and 7 report the median time, the IQR, and the minimum and maximum runtimes required for LScale compilation pass. I use the interquartile range to quantify variations in execution time since it excludes outlier executions.

LGraph **Performance**: The median time required to synthesize a circuit is 0.12-0.57 seconds. The performance of the LGraph pass is relatively stable for most benchmarks. For all benchmarks except the pend benchmark, the IQRfor the execution times range from 0.01-0.06 seconds. The pend benchmark has an IQRof 1.19. The worst-case synthesis time varies significantly across benchmarks. The maximum time required to synthesize a circuit is 1.49-31.84 seconds. Figure 10.5 presents a

breakdown of the `LGraph` pass runtime results by compilation pass.

- **Fragment Synthesis**: The fragment synthesis pass takes between 0.21-2.25 seconds depending on the benchmark. Generally, the time required to synthesize fragments increases with the number of variables and the complexity of the variable relations. I do not present a median, minimum, or maximum execution time for the fragment synthesis pass. I omit these metrics because the compiler generates all synthesized circuits from a single invocation of the fragment synthesis pass.

- **Assembly**: The performance of the assembly pass is relatively stable and does not vary greatly across executions. The median runtime of the assembly pass is between 0.06-0.08 seconds depending on the benchmark. The IQR of the assembly pass runtimes is negligible. The maximum execution time of the assembly pass ranges from 0.06-0.08 seconds.

- **Place+Route**: The worst-case executions of the `LGraph` pass spend the majority of the time in the place+route procedure. The median runtime of the place+route procedure is between 0.31-0.57 seconds depending on the benchmark application. The IQR of the place+route pass runtimes is between 0.00-0.06 seconds for all benchmarks except the `pend` benchmark. Note that the `pend` benchmark has an IQR of 1.19. The uneven performance for the `pend` benchmark arises from performance instabilities in the place+route procedure. The worst-case execution time for the place+route procedure is between 1.06-29.60 seconds.

  These runtime variations result from inefficiencies in the ILP solver. In our experience, `LGraph` must solve a fairly difficult placement and routing problem to map these benchmarks to the analog hardware. This problem is still difficult to solve even with the decomposition employed by the compiler.

`LScale` **Performance**: The performance of the `LScale` compilation pass is relatively stable. The median `LScale` runtime of the `LScale` pass ranges from 0.13-6.13 seconds depending on the benchmark. The IQR is 0.16-2.16 seconds and the maximum execution time is 0.28-5.07 seconds depending on the benchmark.

10.5 Optimality of Scaled and Unscaled ADPs

How optimal are the scaled and unscaled circuits?

I next investigate the optimality of the scaled and unscaled ADPs produced by the compiler. In this context, the optimality of a ADP is determined by three criteria:

- *Does the* ADP *use the minimum number of blocks required to implement the computation?* It is often desirable to minimize the block usage of a produced ADP. If the compiler can efficiently use blocks, it can potentially fit larger computations onto the device. Using fewer blocks can also improve the error characteristics of the produced results in some situations. Unnecessarily introducing analog blocks introduces more sources of error and noise into the computation.

  Section 10.5.2 investigates the distribution of blocks for each of the benchmark applications and discusses the optimality of the produced unscaled ADPs. I discuss cases where the ADP uses additional compute blocks to implement the dynamical system and provide justification for assembly and routing block usages. I can validate that, for all applications, the produced ADPs use the minimum number of routing and assembly blocks to implement the target dynamical system. The compiler also uses the minimum number of required compute blocks for all compute blocks except the multiplier block. The multiplier blocks are generally used to correct for coefficients introduced by the device.

- *Does the* ADP *reach the maximum attainable execution speed for the computation?* It is often desirable to maximize the execution speed of the computation as it would require less time to execute to completion. Section 10.5.3 investigates the execution speeds of the scaled ADPs.

- *Does the* ADP *maximize the dynamic range of time-varying signals in the* ADP*?* It is often desirable to maximize the dynamic range of time-varying signals as it reduces the effect of noise and quantization error on the overall computation. Note that it is not usually possible to fully utilize the entire operating range at every port in a given ADP. Section 10.5.4 investigates the dynamic range of the time-varying signals in the scaled ADPs.

- *Does the* ADP *maximize the magnitude of the values in the* ADP*?* It is often desirable to use data field values with larger magnitudes to reduce the effect of quantization error. Scaling up data field values may potentially improve the fidelity of the computation. Section 10.5.5 investigates the magnitudes of the data field values in the scaled ADPs.

- *Does the* ADP *minimize the* `balanced` *scale objective function specified to the compiler?*: The compiler is instructed to scale the ADP to minimize a `balanced` objective function which jointly maximizes quality and speed. Section 10.5.6 investigates the spread of `balanced`

objective function values produced by the compiler. I demonstrate that the compiler is able to consistently find circuits which minimize the `balanced` objective function.

- *Does the* ADP *produce a wide variety of optimal/close-to-optimal scaling transforms?*: The compiler may identify multiple candidate scaling transforms which all minimize the `balanced` scale objective function. Section 10.5.6 and Section 10.5.7 investigate the spread of `balanced` scale objective values and analog and digital quality measure values produced by the compiler. This spread of quality measures occurs because the compiler returns ten scaled circuits for each unscaled circuit, each of which implements a distinct scaling transform that minimizes the `balanced` scale objective. This objective function maximizes the minimum analog and digital quality measures of the scaled circuit and the execution speed of the ADP.

10.5.1 Metrics

I introduce new metrics for evaluating the optimality of the scaled ADPs. The speed utilization metric captures how close the scaled circuit's execution speed is to the maximum execution speed. The operating range and value utilization metrics capture how effectively a signal or value capitalizes on the available operating range.

The Speed Utilization Metric

This analysis introduces the speed utilization metric. Given an ADP with an execution speed $\tau = \texttt{time-var} \times tc$ and a maximum speed supported by the circuit τ_{max}, the speed utilization is described with the equation:

$$\tau/\tau_{max} \cdot 100$$

The above formulation computes the ratio of the attained execution speed to the maximum possible speed. Note that because the maximum speed of the hardware depends on the blocks modes, it is separately computed for each ADP in this analysis.

A speed utilization of 100% indicates that the ADP attains the maximum possible execution speed supported by the hardware. In contrast, a speed utilization under 100% indicates the ADP does not fully exploit the capabilities of the hardware. Both of these utilizations abide by the frequency constraints the analog device imposes on the computation.

A speed utilization of over 100% indicates that the ADP evolves too quickly and is therefore unsupported by the hardware. ADPs with execution speeds that exceed 100% may trigger frequency-dependent unwanted behaviors in the hardware. Refer to Chapter 5 for a more detailed discussion of frequency-dependent behavior on analog hardware.

The Operating Range Utilization Metric

This analysis introduces the operating range utilization metric. This metric captures how much of the available operating range is used by a time-varying signal or a constant value. I next describe how this metric is computed for a signal with a signal or value with a dynamic range of $[y_{sig}, y'_{sig}]$ which is subject to the operating range restriction $[y_{op}, y'_{op}]$.

Time-Varying Signals: For signals which may take on a range of values ($\mathbb{R}_{sig} < \mathbb{R}sig'$), the utilization is computed with the following equation:

$$\frac{y'_{sig} - y_{sig}}{y'_{op} - y_{op}} \cdot 100\%$$

The above formula computes the ratio of the spread of signal values to the range of values supported by the port.

Fixed Signals: Fixed signals carry constant values within the device. For the target hardware, the programmable constant data fields produce all the fixed signals in the circuit. For a fixed signal ($\mathbb{R}_{sig} = \mathbb{R}_{sig'}$), the utilization is computed with the following equation:

$$\frac{y_{sig}}{max(|y_{op'}|, |y_{op}|)} \cdot 100\%$$

The above equation computes the ratio of the magnitude of the fixed signal to the magnitude of the maximum value supported by the hardware.

An operating range utilization of 100% makes full use of the operating range of a port or data field. Conversely, an operating range utilization less than 100% under-utilizes the supported dynamic range of a port or data field. Both of these operating range utilization quantities do not violate the operating range constraints of the hardware.

An operating range utilization over 100% uses a larger range of values or a larger value than is supported by a port or data field. These signals violate the operating range limitations imposed by the analog hardware. ADPs containing fixed signals with utilizations greater than 100% typically cannot even be programmed to the device. Because fixed signals are typically implemented with constant data fields, generating a fixed signal with a utilization metric over 100% typically requires a data field to be set to a value outside the supported value range.

Note that the utilization is limited to 95% of the full operating range of a port or data field for this evaluation (Section 10.1.1). Therefore, in this analysis, a signal or value fully utilizes the port or data field if they report 95% utilization.

Operating Range Utilization Statistics: The ADPs of interest contain between 14-75 signals and data field utilization measures – one for each port and data field in the hardware. It is therefore infeasible to directly report all of these individual measures. This analysis summarizes the distribution of utilization measures using the median, IQR, minimum and maximum metrics presented in

benchmark	blocks	conns	mult	int	adc	dac	lut	fan	extout	route	kirch
cos	5	5	0	2	0	0	0	1	1	2	0
cosc	9	10	3	2	0	0	0	2	1	2	1
pend	12	13	3/+1	2	1	1	1	2	1	2	1
spring	24	28	7/+1	4	2	2	2	5	1	3	2
vanderpol	14	17	6/+3	2	0	1	0	3	1	2	2
heatN4X2	18	28	5/+5	4	0	1	0	6	1	2	4
forced	18	22	7/+3	4	0	1	0	4	1	2	2
pid	18	21	7/+3	4	0	1	0	4	1	2	4
kalman	18	21	7/+2	4	0	1	0	4	1	2	2
smmrxn	11	13	4/+1	1	0	2	0	2	1	2	3
gentog	28	32	7/+6	4	3	4	3	3	1	4	4
bont4	22	25	9/+1	5	0	0	0	4	1	4	3

Table 10.6: Summary of ADP connections and blocks. The block counts are broken down by block type. The `route` column reports the number of `cin`, `cout`, `tin`, and `tout` blocks in the ADP. The `kirch` column reports the number of times Kirchhoff's law is used to sum signals in the ADP.

Section 10.1.3. I report these metrics for both the time-varying signals and the fixed signals in the ADP.

10.5.2 Optimality of ADP Circuit Topology

Columns 2-15 of Table 10.6 present the block and connection breakdown for each benchmark. The range of block counts is reported for benchmarks where the number of blocks of a particular type varies across ADPs. For each benchmark and block type, I record the number of extraneous blocks relative to the baseline number of blocks as a second number following a slash. For the `cos`, `cosc`, `pend`, `spring`, and `vander` benchmarks, the baseline is computed from hand-implemented configurations written by the hardware designers [50]. For the remaining benchmarks, the baseline is the minimum number of blocks required to implement each benchmark. I compute this baseline metric by counting the operators in the dynamical system computation. Note that, for these benchmarks, it may be necessary to use more blocks than the reported minimum number of blocks to implement the application in practice.

The analog device configurations produced by `LGraph` have between 5-32 blocks and 5-42 connections. Each ADP uses between 1 and 6 current copiers (`fan`) and between 2-4 routing blocks per configuration. The compiler uses every type of compute, copy, and routing block available on the device. The compiler also uses Kirchhoff's law in 0-4 circuit locations to implement addition over analog currents – this is required for 11 of the 12 benchmarks. I observe there is no variance in the block counts across synthesized ADPs. This is likely because the compiler can exercise the place+route procedure to generate ten different ADPs. I manually verified this by inspecting the produced ADPs for each benchmark.

benchmark	total	cin	cout	tin	tout
cos	2	0	0	0	1
cosc	2	0	0	0	1
pend	2	0	0	0	1
spring	2	0	0	1	2
vanderpol	2	0	0	0	1
heatN4X2	2	0	0	0	1
forced	2	0	0	0	1
pid	2	0	0	0	1
kalman	2	0	0	0	1
smmrxn	2	0	0	0	1
gentog	4	0	0	1	2
bont4	4	0	0	1	2

Table 10.7: Breakdown of ADP route blocks. The `total` column reports the total number of route blocks. The `total` column reports the same block counts as the `route` column of Figure 10.6.

For all benchmarks, `LGraph` also uses the minimum number of copier (`fan`) blocks. Each copier block produces three copies of an input signal. To compute the minimum number of copiers, I counted the number of occurrences of each variable in the DSS for each benchmark and then computed the minimum number of copier blocks required. For example, The forced Vanderpol oscillator uses `X` four times, `Y` twice, and `W` twice. The ADP implementation of this oscillator would require a total of four `fan` blocks to produce multiple copies of the relevant signals. Each compiled benchmark uses exactly the minimum required number of copier blocks.

`LGraph` uses the minimum amount of `dac`, `adc`, and `lut`, and `int` compute blocks. These blocks are typically only inserted when the corresponding operator appears in the DSS. `LGraph` will sometimes insert more multipliers than strictly necessary because some HCDCv2 blocks introduce constant coefficients that need to be compensated for in the ADP. For example, the `LGraph` procedure uses two multipliers to implement `A*B`. This is necessary because the`mult` block computes `0.5*X*Y`. The synthesis procedure inserts a second `mult` block which computes `2*A` to cancel out the `0.5` coefficient.

For all benchmarks except the `spring` benchmark, the compiler used the minimum number of route blocks. Figure 10.7 presents a detailed decomposition of the types of route blocks used. The HCDCv2 requires that a `tout` block always be used to route a signal to the `extout` block. The `spring` benchmark maps most of the computation to tile `(0,0)` and therefore must use an additional `tin` and `tout` tile to route the signal of interest to tile `(0,3)` so that it can be routed to an `extout` block. The `bont` benchmark uses one additional `tin` and `tout` block because it requires more integrators than are available on one tile and therefore had to partition the circuit across two tiles. It partitions the circuit so that only one connection straddles both tiles. This is

	% maximum speed			
benchmark	median	iqr	min	max
cos	47.9	1.7	14.9	49.7
cosc	67.5	47.2	52.0	**100.0**
pend	**100.0**	0.0	**100.0**	**100.0**
spring	49.4	2.6	43.4	54.1
vanderpol	**100.0**	0.0	**100.0**	**100.0**
heatN4X2	**100.0**	0.0	98.8	**100.0**
forced	30.5	67.7	30.4	98.2
pid	50.2	3.9	29.8	94.6
kalman	**100.0**	0.0	**100.0**	**100.0**
smmrxn	98.1	6.4	91.6	**100.0**
gentog	27.5	0.6	27.2	71.9
bont4	**100.0**	0.0	97.1	**100.0**

Table 10.8: Optimality of the scaled ADP execution speeds. Each row reports the distribution of % speed utilizations for the listed benchmark. Benchmarks with a speed utilizations of 100% run at the maximum speed supported by the HCDCv2.

the minimal number of distant connections that can be made for this application.

The `gentog` benchmark also partitions the ADP across two tiles because it requires three `lut` and three `adc` blocks but only two reside within a given tile. It therefore uses an extra `tin` and `tout` block to make one long-distance connection between tiles. This partition only requires one connection to straddle two tiles. The `heat` benchmark is partitioned across two tiles and requires two signals cross tiles. It therefore requires a total of two additional `tin` and `tout` blocks – each additional `tin` and `tout` pair is used to make a single long-distance connection between ports.

10.5.3 Execution Speed Optimality

This analysis studies the relationship between the execution speeds of the scaled ADPs and the maximum supported speed of the device. In this analysis, I am primarily interested in whether the scaled ADPs attain the maximum execution speed supported by the analog hardware. Table 10.8 presents the distribution of speed utilization metrics for the scaled ADPs of each benchmark. A speed utilization of 100% (bolded) reaches the maximum supported speed of the device. The scaled ADP speeds all respect the maximum supported speed of the device. This is the case because the scaling procedure considers the device frequency limitations when scaling the circuit.

The compiler is able to produce scaling transforms that fully exploit the performance of the hardware. For all applications except the `cos`, `forced`, and `gentog` benchmarks,

at least one of the scaled ADPs operate at the maximum possible speed supported by the device. For the `pend`, `vanderpol`, `kalman`, and `bont4` benchmarks, all executions achieve the maximum attainable speed of the hardware. For the `heatN4X2` benchmark, more than half of the executions achieve the maximum attainable speed (median). The compiler attains the maximum speed for many of the executions because the `balanced` scale objective function maximizes the time scale factor `time-var`along with the signal quality measures (AQM and DQM). Note that the maximum execution speed is ADP-specific, as the blocks within the circuit impose additional frequency limitations which affect the maximum speed of the circuit.

The scaled ADPs is not always able to attain the maximum achievable speed for all applications. The `cos`, `spring`,`forced`, and `pid` applications report maximum speed utilizations of 49.7%, 54.1%, 98.2% and 94% respectively. The `LScale` pass is likely unable to fully exploit the maximum supported speed of the hardware for these applications because doing so has a significant effect on the quality measures (AQM and DQM). The execution speed and signal quality are interrelated – as the execution speed increases, the dynamic ranges of the state variable signals decrease (assuming a fixed derivative signal). Because of this phenomenon, increasing the speed can reduce the AQM of the circuit to the point where the scaling transform is no longer optimal.

Note that the maximum attainable execution speed changes depending on the blocks in use and the selected block modes. Note that for the HCDCv2, the frequency limit may change if a multiplier, ADC, or integrator is introduced into the circuit. The frequency limit of the multiplier depends on if the multiplier is multiplying two signals (`(m,*,*)` and `(h,*,*)`) of if the multiplier is scaling a signal (`(x,*,*)`. These decisions are made during the `LGraph` compilation pass. The `LScale` pass, therefore, cannot change the frequency limit by strategically selecting modes for this device. I, therefore, omit any analysis on the optimality of the mode selections when studying the execution speed.

10.5.4 Signal Dynamic Range Optimality

This analysis studies the relationship between signal ranges of the time-varying analog and digital signals in the scaled ADPs and the maximum supported operating ranges of the device. In this analysis, I am primarily interested in whether the scaled ADPs effectively

	% signal utilization			
program	median	iqr	min	max
cos	5.9-60.0	0.0-54.0	1.9-10.0	97.9-**100.0**
cosc	40.8-93.1	16.6-68.5	9.8-79.3	81.6-98.9
pend	47.4-89.3	36.0-47.5	8.8-52.4	88.1-95.0
spring	65.4-92.1	4.6-36.4	6.1-14.0	91.0-**100.0**
vanderpol	53.5-**100.0**	0.0-82.7	10.0-93.0	**100.0**
heatN4X2	50.0	0.0	45.0-47.1	50.0
forced	9.6-91.9	27.1-86.4	4.6-9.2	94.3-**100.0**
pid	35.5-77.4	29.9-66.9	5.2-10.0	**100.0**
kalman	13.1-72.3	6.7-52.9	2.3-6.2	83.9-**100.0**
smmrxn	52.9-62.5	0.0	52.3-62.5	56.7-**100.0**
gentog	47.6-71.3	46.5-66.0	4.3-10.0	93.2-**100.0**
bont4	73.1-90.7	16.2-35.6	6.4-67.1	**100.0**

Table 10.9: Optimality of the scaled ADP operating ranges. Each row reports the distribution of operating range utilizations for the listed benchmark. This table only reports the operating range utilizations for time-varying signals

utilize the available operating ranges present within the device.

Table 10.9 presents the distribution of signal utilization metrics for the time-varying signals for each benchmark. For each statistical measure, I report a range of values for that measure seen across all of the scaled ADPs for that benchmark. For example, the `cos` benchmark reports a median signal utilization of `5.9-60.0` – the median signal utilization metrics of the scaled ADPs range from `5.9` to `60`. For all benchmarks, the scaled ADPs report signal utilization metrics equal to or under 100%. Recall any signal utilizations over 100% violate the operating range constraints of the device.

All of the signal utilization metrics are less than 100% because the `LScale` pass produces scaling transforms that respect all of the operating range constraints. For all scaled ADPs, the scaled signals use, at most, `25.0`% to `100`% of the available signal range. For the `cos`, `spring`, `vanderpol`, `forced`, `pid`, `kalman`, `gentog`, and `bont4` benchmarks, the scaled ADP contains at least one signal which fully exploits the available signal range. This indicates the `LScale` pass is able to scale up signals and values to fully exploit the operating ranges present in the hardware.

For all scaled ADPs, the scaled signals use at least `6.2-93.0%` of the available signal ranges in the best case and at least `1.9-26.2`% of the available signal ranges in the worst case. For all benchmarks except the `cos` benchmark, the signal with the lowest utilization uses at least `4`% of the dynamic range. The best-case minimum signal utilizations are likely the best utilizations attainable for these low-performing signals. The compiler maximizes

analog and digital quality measures, which maximize the SNR of the signals with the smallest dynamic ranges in the circuit. This operation also maximizes the utilization of these signals.

For all benchmarks, there is at least one ADPs that reports a median signal utilization of at least 50%. Of these applications, the `spring`, `vanderpol`, `heatN4X2`, `smmrxn`, and `bont4` benchmarks report a median signal utilization of at least 50% for all ADPs. Therefore, for a significant fraction of the benchmark applications, the compiler can attain a reasonably good dynamic range for the majority of the signals.

The signal utilizations vary significantly across individual scaled ADPs and benchmarks. Depending on the benchmark application and ADP, the median signal utilization ranges from 5.9%-100.0%, and the IQRranges from 0.0-85.7%. This high degree of variation in signal utilizations across benchmarks is expected, as it's not typically possible to maximize the signal's dynamic range on every single wire. The degree to which the compiler can maximize individual signal ranges depends on the characteristics of the circuit implementing the dynamical system.

For all benchmark applications, the minimum, maximum, median, and IQR measures vary significantly across ADPs. The high degree of variation in signal utilizations for ADPs implementing the same benchmark suggests that the compiler produces a wide variety of equally viable scaling transforms, each of which scales the signals very differently. This is explored further in Section 10.5.6 and Section 10.5.7.

10.5.5 Data Field Value Optimality

This analysis presents a statistical summary of the data field value utilization scaled ADPs for each benchmark application. In this analysis, I am primarily interested in whether the scaled ADPs effectively scale the data field values present within the device.

Table 10.10 report the utilization of the scaled data field values. The table reports the minimum and maximum values for each statistical measure. For all benchmarks, the scaled ADPs report value utilizations less than or equal to 100%. All of the utilizations fall under 100% because the `LScale` pass produces scaling transforms that respect all of the operation range constraints.

The maximum utilization for the ADPs range from `6.6`% to `100.0`%. For all applications except the `cos` benchmark, the compiler produces scaled ADPs that maximize at least one

	% value utilization			
program	median	iqr	min	max
cos	6.6-**100.0**	0.0	6.6-**100.0**	6.6-**100.0**
cosc	35.2-**100.0**	18.1-85.7	7.4-16.1	**100.0**
pend	62.9-70.3	5.4-40.1	36.8-57.9	**100.0**
spring	33.8-64.8	56.9-77.0	13.4-20.5	**100.0**
vanderpol	34.7-61.0	18.4-36.7	19.4-33.8	**100.0**
heatN4X2	68.5-73.0	9.4-25.6	50.0	**100.0**
forced	33.0-97.7	12.3-69.8	9.1-36.8	**100.0**
pid	36.2-78.2	21.5-61.1	9.3-41.0	**100.0**
kalman	23.4-80.1	10.0-63.3	16.9-46.1	55.4-**100.0**
smmrxn	21.6-91.6	40.3-56.5	9.5-33.3	**100.0**
gentog	45.4-93.3	19.9-57.4	9.2-47.7	**100.0**
bont4	40.5-**100.0**	5.0-81.7	4.0-40.9	**100.0**

Table 10.10: Optimality of the scaled ADP operating ranges. Each row reports the distribution of operating range utilizations for the listed benchmark. This table only reports the operating range utilizations for fixed signals and constant values.

value. This observation indicates the `LScale` pass can scale up values to attain the maximum possible values supported by the programmable data fields.

For the scaled ADPs, the best-case minimum value utilization is between `9.2-100.0`% and the worst-case value utilization is between `1.9-26.2`%, depending on the benchmark. The best-case minimum value utilizations are likely the best utilizations attainable for these low-performing data field values. Because the compiler maximizes the SNR of the smallest amplitude values in the circuit, the utilization of these values is likely also maximized by the compiler.

For all benchmarks, there is at least one ADP which reports a median value utilization of at least 50%. Of these applications, the `pend` and `heatN4X2` benchmarks report a median value utilization of at least 50% of the values for all ADPs. Therefore, for a significant fraction of the benchmark applications, the compiler can identify an ADP which attains a reasonably good magnitude for the majority of the values. A value utilization greater than 50% translates to at most 1.56% quantization error.

The utilization distribution metrics also vary widely across scaled ADPs and benchmarks. Depending on the benchmark application, the median value utilization is between 6.2-94.3%, and the value utilization IQR is between 0%-80.7%. This observation suggests that the constant data field values take on a broad range of values within each benchmark application. The high degree of variation in signal utilizations for ADPs implementing the same benchmark suggests that the compiler produces a wide variety of equally viable scaling

	`balanced` scale objective				
program	median	iqr	min	max	worst-case
cos	9.04e-07	6.16e-07	5.96e-07	8.96e-05	1.26e-04
cosc	1.03e-06	9.13e-07	3.44e-07	2.18e-06	2.68e-01
pend	3.47e-05	1.13e-05	2.33e-05	1.01e-04	3.48e-01
spring	1.69e-04	1.20e-05	1.34e-04	5.25e-04	1.10e+04
vanderpol	1.96e-07	3.74e-08	1.21e-07	7.42e-07	4.97e+03
heatN4X2	1.07e-07	9.85e-09	1.05e-07	1.16e-07	2.37e+02
forced	5.71e-06	7.32e-06	2.00e-06	1.41e-05	1.01e+00
pid	3.12e-06	1.98e-07	8.61e-07	5.15e-06	1.01e+15
kalsmooth	2.86e-06	2.16e-06	2.10e-06	7.60e-06	1.14e+18
smmrxn	1.29e-06	2.24e-06	8.81e-07	4.29e-06	1.30e+03
gentog	3.94e-04	1.55e-03	3.13e-04	2.22e-03	3.26e+07
bont4	8.44e-06	8.41e-06	6.41e-06	1.15e-04	2.50e+02

Table 10.11: Distribution of `balanced` scale objective values for the benchmark applications. Lower scale objective values are better. The worst-case `balanced` scale objective values are obtained by maximizing the objective function.

transforms, each of which scales the data field values in the ADP very differently. This is explored further in Section 10.5.6 and Section 10.5.7.

10.5.6 `balanced` Scale Objective Function Value Optimality

Table 10.11 presents the objective function values for the scaled ADPs. Columns 2-5 present the distribution of `balanced` objective function values and Column 6 presents the maximum (worst case) possible value for `balanced` the objective function. I computed the maximum possible value by maximizing (instead of minimizing) the `balanced` objective function for each of the benchmark applications. The worst-case values range from `1.26e-4` to `1.14e+18`, depending on the benchmark.

The scaled ADPs are guaranteed to be optimal since the underlying optimization problem is convex. However, Table 10.11 reports a distribution of values. These variances occur because the compiler produces multiple scaled ADPs by progressively removing already generated scaling transforms and mode selections from the solution space. The compiler, therefore, generates a sequence of scaling transforms of decreasing optimality, each of which is optimal given the constraints over the solution space. Table 10.11 are the smallest attainable values for the benchmarks.

For all of the benchmark applications, the median values are several orders of magnitude smaller than the worst-case values. The median objective function values range from `1.07e-7` to `3.94e-4`, depending on the benchmark. This observation indicates that the

program	AQM	DQM	time-varying signal qualities			
			label	state vars	observe	digital
cos	3.6-19.0	8.0-121.1	11.4-57.0	11.4-114.0	58.9-114.0	
cosc	46.5-103.2	9.0-19.6	46.5-103.2	82.2-180.9	76.1-106.2	
pend	41.9-99.5	44.9-70.7	41.9-99.5	90.0-99.5	92.9-101.8	
spring	5.8-13.3	16.3-25.0	47.5-95.0	76.8-128.6	87.8-100.4	52.1-87.2
vanderpol	47.5-95.0	23.7-41.2	47.5-95.0	101.6-190.0	101.1-106.1	
heatN4X2	47.5-52.9	65.7-66.5	47.5-52.9	95.0-95.0	51.3-53.7	
forced	9.5-17.5	11.1-44.9	18.2-55.8	18.2-111.6	67.5-114.0	
pid	7.9-12.8	11.3-50.1	45.9-95.0	45.9-99.9	94.4-114.0	
kalman	2.8-7.0	20.6-56.3	13.6-52.6	25.0-63.5	87.6-114.0	
smmrxn	5.2-13.7	11.6-40.3	50.3-59.4	100.6-118.7	59.7-114.0	
gentog	6.1-7.9	11.2-63.0	52.0-84.2	52.0-84.2	95.9-100.3	81.8-120.6
bont4	1.2-1.6	4.9-49.9	12.1-106.1	12.1-127.5	85.0-106.1	

Table 10.12: Breakdown of analog quality measures for scaled ADPs

space of candidate scaling transforms that satisfy all the constraints contains both good and bad scaling transforms. This observation also indicates that the compiler can effectively navigate the space of candidate scaling transforms and identify good transforms.

For all benchmark applications, the IQR of the scale objective values falls between 9.85e-09 and 1.55e-03. These value spreads are relatively narrow, especially when compared to the worst-case objective function value. This observation indicates that there are multiple good candidate scaling transforms that have similar scale objectives. The compiler is able to identify these scaling transforms and return multiple scaled ADPs with comparable optimality.

10.5.7 Analog and Digital Quality Measure Breakdown

I next study the spread of analog quality measures (AQMs) and digital quality measures (DQMs) for the scaled ADPs. The AQMs capture the ratio of the maximum value of the scaled analog signals to the noise floor. For example, a scaled signal with an AQM of 3.5 has a maximum signal amplitude which is 3.5x the noise floor of the wire carrying the signal. The DQMs capture the ratio of the maximum value of the scaled digital signals to the quantization error. For example, a scaled data field value with a DQM of 10.5 implements a value that is 10.5x the quantization error of the data field. This scaled value would therefore have $10.5^{-1}\times$ 100%=9.5% error. Generally speaking, signals and values with larger AQMs or DQMs are less affected by analog noise and error.

Table 10.12 presents a breakdown of the analog and digital quality measures for the

scaled ADPs. The `balanced` scale objective function jointly maximizes the execution speed and all of the quality measures presented in this table. Columns 2 and 3 present the range of the minimum AQMs and DQMs across all ADPs. Columns 4-7 present a breakdown of the smallest quality measures for time-varying signals, organized by signal type:

- `label` (Column 4): This column reports the range of minimum quality measures for the subset of signals which have been affixed with ADP `source` source labels. These signals implement variables and dynamical system expressions. These quality measures may be analog quality measures or digital quality measures since time-varying digital signals can also implement dynamical system expressions.

- `state vars` (Column 5): This column reports the range of minimum analog quality measures for the subset of signals which implement dynamical system state variables.

- `observe` (Column 6): This column reports the range of minimum analog quality measures for the subset of signals which are externally accessible and can be measured with an external measurement device.

- `digital` (Column 7): This column reports the range of minimum digital quality measures for the digital time-varying signals.

Overall, both the DQMs and AQMs vary substantially within each benchmark. These variances occur because there exist multiple viable scaling transforms with different quality measure characteristics that resolve to approximately the same scale objective function value. The compiler, therefore, identifies multiple close-to-optimal scaling transforms with different quality characteristics.

Depending on the benchmark, the DQMs range from 4.92-70.73. The digital value with the smallest reported minimum DQM is 4.92x the quantization error of the data field or digital port. This translates to about 20% error for the value with the lowest DQM.

Depending on the benchmark, the AQMs range from 1.2-103.25. The maximum value for the scaled signal with the smallest AQM is 1.24x the noise floor. The maximum value for the scaled signal with the largest minimum AQM is 103.25x the noise floor. The `bont4` benchmark reports the smallest minimum overall AQM of 1.2-1.6. Note that an ADP may still accurately execute a computation even if it contains signals with small dynamic ranges,

provided these signals do not have a significant impact on the overall computation. Refer to Section 10.8.2 for a correlation study of the quality measures and the end-to-end result.

For all benchmark applications, the scaled ADPs report higher minimum AQMs for the signals implementing observations, state variables, and the signals affixed with `source`labels. This is likely because the `balanced` scale objective individually maximizes these minimum AQMs in addition to the overall minimum AQM of the circuit.

The scaled ADPs also report much higher minimum DQMs for time-varying digital signals. Depending on the benchmark, the minimum DQMs is between 52.1 and 120.6. These DQMs are much higher than the overall minimum DQM because the `balanced` calibration strategy independently maximizes the minimum quality measure for these signals.

10.6 Viability of Unscaled ADPs

Why is the scaling transform necessary for successful compilation?

I next investigate why the unscaled ADPs are not amenable to execution. This analysis justifies the need for a scaling transform comprised of time and magnitude scale factors. I demonstrate that the unscaled ADPs violate the operating range and frequency constraints hardware in this analysis. In this analysis, I study the execution speed, data field values, and signal dynamic ranges in the unscaled ADPs. This analysis makes use of the analysis metrics introduced in Section 10.5.1.

10.6.1 Execution Speed of Unscaled ADPs

First, I investigate whether the time scaling component of the scaling transform is necessary for producing ADPs that can viably run on the analog hardware. Table 10.13 presents the distribution of speed utilization metrics for the unscaled ADPs of each benchmark. Each unscaled ADP directly maps simulation time to hardware time (the time scale factor τ is 1.0).

For all applications, the unscaled ADP speeds all exceed the maximum supported speed of the device. These unscaled ADPs, therefore, cannot be safely run on the analog hardware. The compiler must therefore adjust the execution speed of all of the unscaled ADPs to respect the frequency limitations on the device.

		% value utilization			
program	median	iqr	min	max	
cos	unsc	157.5	0.0	157.5	157.5
cosc	unsc	157.5	0.0	157.5	157.5
pend	unsc	315.0	0.0	315.0	315.0
spring	unsc	315.0	0.0	315.0	315.0
vanderpol	unsc	315.0	0.0	315.0	315.0
heatN4X2	unsc	157.5	0.0	157.5	157.5
forced	unsc	315.0	0.0	315.0	315.0
pid	unsc	157.5	0.0	157.5	157.5
kalman	unsc	315.0	0.0	315.0	315.0
smmrxn	unsc	315.0	0.0	315.0	315.0
gentog	unsc	315.0	0.0	315.0	315.0
bont4	unsc	157.5	0.0	157.5	157.5

Table 10.13: Execution speed utilization of unscaled ADPs for each benchmark application. Execution speed utilizations exceeding 100% violate the frequency constraints of the HCDCv2.

10.6.2 Signal Dynamic Ranges of Unscaled ADPs

Table 10.14 presents the operating range utilization metrics for the time-varying signals in the ADPs. All utilizations greater than 100% violate the operating range constraints of the hardware.

The `cosc`, `pend`, `spring`, `vanderpol`, `heatN4X2`, `forced`, `pid`, and `gentog` benchmarks all contain signals with dynamic ranges which exceed the operating range supported by the analog hardware. The unscaled ADPs for these benchmark applications therefore contain signals which saturate ports within the HCDCv2. These applications therefore cannot be safely run on the analog hardware.

The `cos`, `kalman`, and `bont4` benchmarks all contain signals with dynamic ranges which fall within the maximum supported operating ranges. However, these benchmarks still cannot be executed on the analog hardware as they violate the execution speed constraints of the device.

	% signal utilization			
program	median	iqr	min	max
cos	5.3-52.6	0.0-47.4	5.3	87.7
cosc	65.8-657.9	13.2-460.5	52.6-526.3	877.1
pend	7.9-78.9	0.0-71.1	7.9-78.9	131.6
spring	7.9-78.9	14.5-71.1	3.9-6.4	193.0
vanderpol	131.6	0.0-118.4	13.2-131.6	219.3
heatN4X2	10.5-105.3	0.0	10.5-105.3	175.4
forced	11.6-115.8	3.2-110.5	5.3	193.0
pid	10.5-105.3	0.0-94.7	2.6-17.5	175.4
kalman	3.7-36.8	2.6-50.0	0.7	87.7
smmrxn	2.6-26.3	0.0	2.6-26.3	43.9
gentog	52.6	77.9-835.8	2.6	1403.5
bont4	0.7-4.2	4.0-9.6	0.0	5.3-52.6

Table 10.14: Distribution of operating range utilizations for the time-varying signals in the unscaled ADPs. Each row reports the distribution of operating range utilizations for each benchmark application. Only the operating range utilizations for time-varying signals are reported. Operating range utilizations exceeding 100% violate the operating range constraints of the HCDCv2.

10.6.3 Data Field Values of Unscaled ADPs

Table 10.14 presents the operating range utilization metrics for the data field values in the ADPs. All values with utilizations greater than 100% violate the operating range constraints of the data fields and cannot be written to the device.

For all applications except the `cos` application, the unscaled ADPs contain value utilizations that exceed 100%. Therefore, almost all of the unscaled ADPs cannot even be written to the target device. For these applications, the compiler must scale the signal and value magnitudes so that they can be written to the analog device.

For the `cos` application, all the data field values have operating range utilizations less than or equal to 53%. The unscaled ADPs for the `cos` application can therefore be written to the device. However, the `cos` application cannot be executed on the device since it exceeds the maximum execution speed of the device.

	% value utilization			
program	median	iqr	min	max
cos	53.0	0.0	53.0	53.0
cosc	105.3	88.4	23.2	477.4
pend	53.0	53.4	18.9	176.8
spring	53.0	52.6	15.8	176.8
vanderpol	84.2	81.9	21.1	176.8
heatN4X2	106.1	0.0	10.5-105.3	176.8
forced	176.8	421.1	26.3	530.5
pid	79.2	106.4	26.3	842.1
kalman	45.1	153.5-160.3	3.2-4.2	212.2
smmrxn	31.6-42.1	80.6-94.3	2.6-26.3	212.2
gentog	105.3	79.6	2.6	176.8
bont4	6.1	13.3	1.4	176.8

Table 10.15: Distribution of operating range utilizations for the fixed signals/data field values in the unscaled ADPs. Each row reports the distribution of operating range utilizations for each benchmark application. Only the operating range utilizations for fixed signals and data field values are reported. Operating range utilizations exceeding 100% violate the operating range constraints of the HCDCv2.

10.7 Scaling Transform Complexity

Are the scaling transforms complex enough to warrant an automated approach?

This section provides a general breakdown of the scaled circuit. I summarize the range of time and magnitude scale factors and identify which proportion of the scale factors are unique. This analysis demonstrates that the compiler derives complex scaling transforms with many distinct scale factors which span a wide range of values.

Table 10.16 presents the time and magnitude scale factor statistics and the statistics of the injected variables. Column 2 presents the range of time scale factor values. Columns 3-5 present the total number of magnitude scale factors, the number of unique magnitude scale factor values, and the range of magnitude scale factor values. Columns 6-9 present the total number of injected variables, the number of unique injected variable values, and the range of injected variable values.

The scaling transformations produced by `LScale` have between 4 and 30 unique signal scaling factors. For the `pend`, `spring`, and `gentoggle` benchmarks, `LScale` injects 2-6 unique constant coefficients into the expression data fields to more freely scale the circuit. The magnitude scale factors range from 0.01-34648.75 the injected values range from 0.53-23.56. The `bont4` benchmark reports a high scaling factor for the signal implementing the expression `0.013*transB`. This signal is scaled up because the unscaled dynamic range for

		magnitude scale factors			injected values		
benchmark	tau	total	unique	range	total	unique	range
cos	0.09-0.32	14	4	0.12-3.73	0	0	
cosc	0.33-0.63	30	10	0.06-4.32	0	0	
pend	0.32	39	13	0.39-23.56	2	2	0.80-23.56
spring	0.14-0.17	82	23-24	0.15-15.39	4	4	1.08-15.39
vanderpol	0.32	49	14-15	0.18-7.60	0	0	
heatN4X2	0.63-0.63	63	17	0.26-4.75	0	0	
forced	0.10-0.31	63	19-20	0.06-8.64	0	0	
pid	0.19-0.60	63	20-21	0.05-12.67	0	0	
kalman	0.32	63	21	0.08-27.46	0	0	
smmrxn	0.29-0.32	37	11	0.09-21.55	0	0	
gentog	0.09-0.23	90	30	0.01-18.85	6	6	0.53-18.85
bont4	0.62-0.63	75	26	0.57-34648.75	0	0	

Table 10.16: Summary of scaling transform magnitude scale factors, time scale factors, and injected coefficients. The `total` column reports the total number of scale factors/injected coefficients, the `unique` column reports the number of unique scale factors/injected values, and the `range` column reports the range of values for the scale factors and injected values.

this term is very small.

The time scaling factors configure the ADPs to execute at 0.09x-0.63x the baseline speed of the analog device (126000 Hz). The slowest simulation (0.09x baseline) executes one unit of simulation time in $7.00 \cdot 10^{-5}$ seconds of wall-clock time.

10.8 Compilation Outcomes and Result Quality

Section 10.5.6 reported that the scaled ADPs for a given benchmark report comparable `balanced` scale objective values. However, in practice, the produced waveforms vary significantly depending on the execution. This disparity between the static measure of circuit optimality and the actual end-to-end result quality may arise because the compilation procedure fails to consider the full range of hardware behaviors that may impact the end-to-end result.

I present an analysis that investigates the relationship between the quality of the end-to-end result and the scaled ADP characteristics. The results of this analysis can be used to inform future optimizations to the compiler. For each investigated ADP characteristic, I study if the chosen calibration strategy affects the relationship between the target ADP characteristic and the end-to-end result. If the calibration algorithm can eliminate an unwanted low-level behavior that is not currently considered by the compiler, then a novel

compiler optimization may not be required. I study the effect of the following ADP characteristics on end-to-end result quality and evaluate if modifying the calibration strategy reduces the impact of some of these unmodeled effects:

- **Block Instance Selections (Section 10.8.1)**: I investigate the relationship between the locations of the blocks in the scaled ADPs and the end-to-end result.

- **Scale Objective Function Values**: I investigate the relationship between the `balanced` scale objective function values and the end-to-end result quality. The goal of this analysis is to understand how predictive the b`balanced` scale objective function is of the end-to-end result. I also perform a more detailed analysis of how the analog and digital quality measures and execution speed are related to the end-to-end result quality. The goal of this analysis is to determine if the `balanced` scale objective could instead be replaced with a simple objective function that maximizes a single quality measure or the execution speed.

Metrics and Methodology

The block instance analysis studies the `maxfit` and `minerr` executions from Section 10.1. The analysis groups the `minerr` and `maxfit` executions by originating circuit number (0-10). Each circuit number corresponds to an unscaled ADP which was generated by the `LGraph` compilation pass. I manually inspected each unscaled ADP manually to ensure that only the block locations differ across generated circuits.

The quality measure analysis and dynamic signal range analyses perform an in-depth analysis of the first unscaled ADP generated by the `LGraph` pass. I chose to focus on a single unscaled ADP to control for the effect of different block instance selections on the end-to-end result. These analyses make use of the `single` scaling objective function, a new scaling objective function that produces a wide variety of scaled circuits:

- `single`: The ADPs scaled with the `single` objective maximize one magnitude scale factor or time scale factor. The compiler produces a `single` ADP for each time and magnitude scale factor in the ADP.

  Each `single` ADP must have a minimum AQM of 1.0, a minimum DQM of 5.0 for data field values, and a minimum DQM of 5.0 for time-varying digital signals. The

minimum time scale factor value is 0.001. These restrictions over the signal quality and speed prevent the compiler from producing scaling transforms that completely degrade the quality of any single signal. I intentionally select a loose lower bound on the minimum AQM and DQM and a loose lower bound for the execution speed. These loose minimum bounds are smaller than all the observed speeds, AQMs, and DQMs reported in Sections 10.7 and 10.5.7.

The analyses presented in this section work with four types of executions. These executions capture all scaling objective and calibration strategy combinations:

- `single/maxfit` - The compiler uses the `single` scaling objective and targets blocks calibrated with the `maximize_fit` calibration strategy. The compiler produces a scaled ADP for each time and magnitude scale factor in the ADP.

- `single/minerr` - The compiler uses the `single` scaling objective and targets blocks calibrated with the `minimize_error` calibration strategy. The compiler produces a scaled ADP for each time and magnitude scale factor in the ADP.

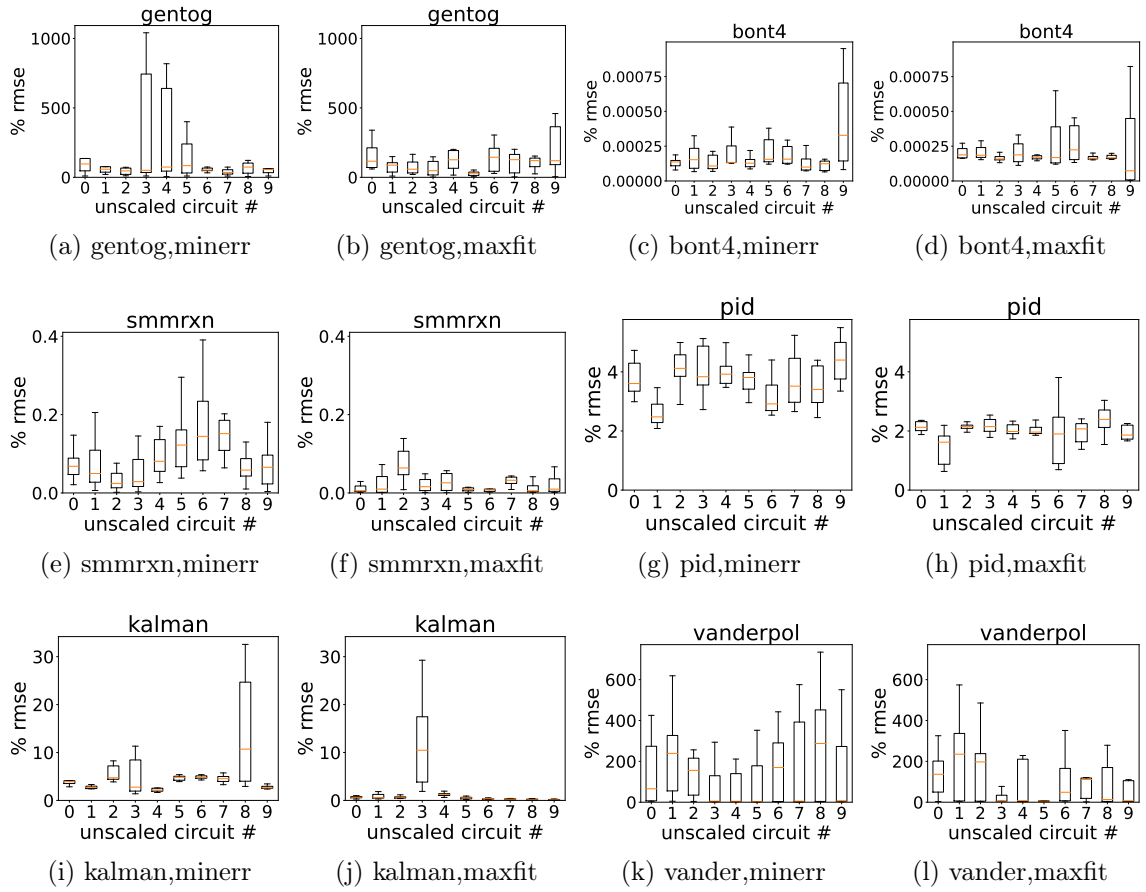
- `bal/maxfit` - The compiler uses the `balanced` scaling objective and targets blocks calibrated with the `maximize_fit` calibration strategy. The compiler produces 10 scaled ADPs.

- `bal/minerr` - The compiler uses the `balanced` scaling objective and targets blocks calibrated with the `minimize_error` calibration strategy. The compiler produces 10 scaled ADPs.

10.8.1 Block Instance Selection and Result Quality

Does the selection of block instances in the ADP *have a significant effect on the end to end error?*

Figure 10-7 and Figure 10-8 present the distribution of % rmses (y axis) for each generated `LGraph` circuit (x axis). Each plot breaks down the % rmse by circuit number for either the `minerr` or `maxfit` executions of a particular benchmark. Each box and whisker plot within the graph reports the distribution of errors for the `LScale` circuits generated



(a) gentog,minerr (b) gentog,maxfit (c) bont4,minerr (d) bont4,maxfit

(e) smmrxn,minerr (f) smmrxn,maxfit (g) pid,minerr (h) pid,maxfit

(i) kalman,minerr (j) kalman,maxfit (k) vander,minerr (l) vander,maxfit

Figure 10-7: Breakdown of % rmses by originating unscaled ADP. Each plot reports the breakdown for the `minerr` executions or for the `maxfit` executions of a benchmark. The y-axis reports the % rmse and the x-axis reports the identifier of the originating unscaled ADP. Refer to Section 10.1.3 for an overview of box plots.

from the target `LGraph` circuit with the specified calibration strategy (`maximize_fit` or minimize_error).

For all the benchmarks, the distribution of % rmses varies significantly across `LGraph` circuits. This observation indicates that the chosen block instances have a significant impact on the end-to-end result of the computation. This may be because different instances of the same block have different error characteristics; these differences adversely impact the fidelity of the result.

The `maxfit` executions more consistently produce good results across diffrent `LGraph` circuits than the `minerr` executions. For the `gentog`, `pid`, `smmrxn`, `forced`, `spring`, and `cos` applications, the `maxfit` results report significantly less error (lower median) and deliver low-error results more consistantly (smaller IQR) than the `minerr` results. For these applications,

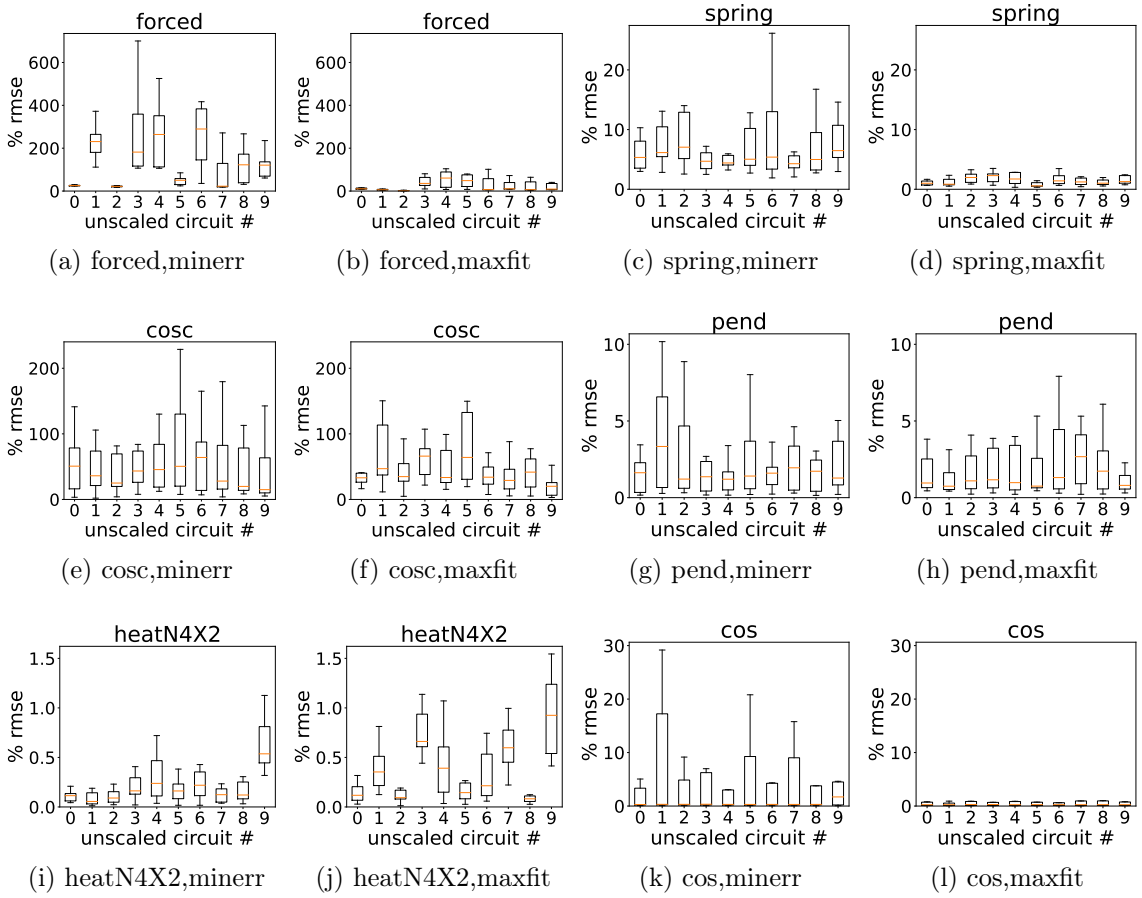


Figure 10-8: Breakdown of % rmses by originating unscaled ADP. Each plot reports the breakdown for the `minerr` executions or for the `maxfit` executions of a benchmark. The y-axis reports the % rmse and the x-axis reports the identifier of the originating unscaled ADP. Refer to Section 10.1.3 for an overview of box plots.

the scaled ADPs which target the `maximize_fit` calibration strategy definitively produces better results than the `minerr` executions. One possible explanation is the `maximize_fit` calibration routine is able to eliminate unwanted behavior in a broader range of blocks since it allows for calibrated blocks to have non-unity gains.

In some cases, the `minerr` executions outperform the `maxfit` executions. For the `kalman`, `vanderpol`,`cosc`, and `pend` executions, the `maxfit` executions report a higher error (higher median) for some `LGraph` circuits and lower error (lower median) for others. For these applications, the best calibration strategy to use differs depending on the circuit. One possible explanation for this is that in some circuits, the `minimize_error` calibration routine is able to eliminate unwanted behavior and also deliver unity gain.

program	calib strategy	speed	DQM	AQM	time varying signals: label	state vars	observe	digital	scaling objective
cos	`minerr`	**-0.69**	0.19	-0.04	0.02	**0.59**	0.22		-0.06
cos	`maxfit`	0.18	0.09	0.02	0.17	-0.14	0.15		-0.15
cosc	`minerr`	0.06	-0.22	-0.21	-0.38	-0.39	**-0.99**		**0.99**
cosc	`maxfit`	0.09	-0.07	-0.33	-0.26	-0.20	-0.40		**1.00**
pend	`minerr`	0.00	-0.24	-0.22	-0.35	-0.11	**-0.69**	-0.03	**0.96**
pend	`maxfit`	0.00	-0.43	-0.46	-0.46	-0.17	-0.35	-0.38	**0.59**
spring	`minerr`	0.04	-0.09	-0.04	-0.09	-0.08	**-0.69**	-0.34	**0.61**
spring	`maxfit`	0.17	0.10	-0.14	-0.21	-0.19	-0.49	-0.15	0.42
vanderpol	`minerr`	n/a	-0.22	-0.27	-0.35	-0.32	-0.26		**0.74**
vanderpol	`maxfit`	n/a	-0.41	-0.43	-0.40	-0.31	-0.38		0.43
heatN4X2	`minerr`	-0.00	-0.20	-0.31	-0.35	-0.25	0.16		0.34
heatN4X2	`maxfit`	-0.00	-0.27	-0.22	-0.41	0.05	**-0.58**		**0.99**
forced	`minerr`	n/a	-0.16	-0.44	-0.48	0.05	**-0.52**		**0.73**
forced	`maxfit`	0.00	-0.23	-0.29	-0.43	0.04	-0.36		**0.77**
pid	`minerr`	0.44	-0.47	**-0.63**	**-0.61**	-0.45	**-0.64**		**0.88**
pid	`maxfit`	0.28	-0.41	-0.49	-0.44	-0.28	**-0.57**		**0.97**
kalman	`minerr`	0.00	-0.22	-0.29	-0.27	0.03	-0.35		**0.80**
kalman	`maxfit`	-0.00	**-0.74**	-0.30	-0.18	0.17	0.06		0.11
smmrxn	`minerr`	0.20	-0.20	-0.21	**-0.72**	0.23	**-0.79**		**0.98**
smmrxn	`maxfit`	0.07	-0.04	-0.33	-0.43	0.06	-0.44		**1.00**
gentog	`minerr`	0.06	-0.03	-0.13	-0.09	-0.02	**-0.62**	-0.06	**0.99**
gentog	`maxfit`	0.14	-0.19	-0.04	-0.16	-0.14	**-0.65**	-0.14	**0.93**
bont4	`minerr`	n/a	-0.11	-0.49	-0.38	**-0.51**	-0.49		**0.62**
bont4	`maxfit`	n/a	-0.12	-0.23	-0.45	-0.46	**-0.51**		**0.96**

Table 10.17: Relationship between ADP execution speed, quality measures, and `balanced` scale objective function value and the end-to-end result. Each cell reports the Pearson correlation coefficient (PCC) between the % rmse and the target ADP characteristic. ADP characteristics may be weakly correlated/uncorrelated (regular), correlated (**bold**) or strongly correlated (**blue, bold**) with the end-to-end error. All cases where the PCC cannot be computed are marked as not applicable (`n/a`).

Conclusion: Neither calibration strategy fully attenuates away the errors resulting from variations in the selected block instances. Furthermore, block instance selection is a significant source of unpredictability in the end-to-end result. For this reason, It would be productive to invest time in developing analyses that relate the block instance-specific characteristics to the fidelity of the end-to-end result.

10.8.2 The Scale Objective Function and Result Quality

I next investigate the relationship between the ADP characteristics and the end-to-end result. In this section, I study the correlation between the end-to-end result and the ADP quality measures, execution speed, and `balanced` scale objective function values. The goal

of this analysis is to identify which scaled ADP characteristics are strongly predictive of the end-to-end result. I use the Pearson correlation coefficient to study the relationship between each ADP characteristic and the end-to-end result. The Pearson correlation coefficient (PCC) is a measure of linear correlation between two sets of data. The PCC may be any value from -1 to 1. A value close to -1 indicates that the error decreases as the ADP characteristic increases. A value close to one indicates that the error increases as the ADP characteristic increases. A value close to zero indicates that the ADP characteristic has little to no effect on the quality of the end-to-end result. This analysis describes PCCs with a magnitude above 0.5 as correlated and PCCs with a magnitude above 0.9 as strongly correlated. Note that the PCC cannot be computed for ADP characteristics which take on the same value for all executions.

Table 10.17 presents the Pearson correlation coefficients for the execution speed, quality measures, and `balanced` scale objective values of the scaled ADPs. Each PCC reports how strongly correlated the ADP characteristic is with the end-to-end result. For example, for the `cosc` executions which target the `maximize-fit` calibration strategy (row 4), the `balanced` scale objective value is strongly correlated (has a PCC of `1.00`) with the error of the end-to-end result. For this benchmark, minimizing the `balanced` scale objective is therefore likely to also minimize the error of the end-to-end result.

Columns 1 and 2 report the benchmark and calibration strategy of each entry in the table. The `minerr` calibration strategy entries report the correlations for the the `bal/minerr` and `single/minerr` executions. The `maxfit` calibration strategy entries report the correlations for the `bal/maxfit` and `single/maxfit` executions.

Columns 3, 4, and 5 present the correlations for the execution speed and the minimum DQM and AQM. Columns 6-9 present the PCCs between the end-to-end result and the minimum quality measures for different subsets of time-varying signals in the ADP. I previously introduced these classes of time-varying in Section 10.5.7:

- `label` (Column 6): This column reports the quality measure correlations for the subset of signals which have been affixed with ADP `source` source labels. These signals implement variables and dynamical system expressions.

- `state vars` (Column 7): This column reports the quality measure correlations for the subset of signals which implement dynamical system state variables.

- `observe` (Column 8): This column reports the quality measure correlations for the subset of signals which are measured in this evaluation.

- `digital` (Column9): This column reports the quality measure correlations for the subset of digital time-varying signals.

Column 10 presents the correlation between the end-to-end result and the `balanced` scale objective function value.

Columns 5-8 presents the correlations for the minimum AQMs for analog signals with source labels, the analog signals implementing state variables, and the observed analog signals, respectively. Column 9 presents the correlation for the minimum DQM for all time-varying signals within the ADP. Column 10 presents the correlation of the `balanced` scaling objective function value and the end-to-end error.

Scaling Objective:For all benchmarks except the `cos` benchmark, the `balanced` scaling objective value are correlated with the end-to-end result. Fhe `cosc`, `pend`, `heatN4X2`, `pid`, `smmrxn`, `gentog`, and `bont4` benchmarks, the end-to-end result quality is strongly correlated with the `balanced` scaling objective value. This indicates that, in many cases, the `balanced` scaling objective value is strongly predictive of the quality of the end-to-end result.

The `cos` benchmark quality is not strongly correlated with the `balanced` scaling objective function value. For this benchmark, the execution speed is negatively correlated with the end-to-end error, and the state variable AQM is positively correlated with the end-to-end error. These two correlated values likely work against each other in the `balanced` scaling objective function. For this application, it would likely be better to directly maximize the speed, subject to a set of minimum AQM constraints.

For the `spring`, `vanderpol`, and `kalman` benchmark applications, only the executions which target the `minimize_error` calibration strategy have `balanced` scaling objective values which strongly correlate with the end-to-end result. This indicates that the `balanced` scaling objective value doesn't identify low-error scaled ADPs as effectively when targeting the `maximize_fit` scaling objective. This may be because other factors, such as unused portions of the signal ranges, and not accounted for in the objective function.

For the `heatN4X2` benchmark, only the executions which target the `maximize_fit` calibration strategy have `balanced` scaling objective values which strongly correlate with the end-to-end result. One possible explanation is that the `minimize_error` calibration strategy

introduces unmodelled block behaviors into the ADP which the compiler cannot compensate for in compilation.

Speed and Quality Measures: The `cosc`, `pend`, `spring`, `heatN4X2`, `forced`, `pid`, `smmrxn`, `gentog`, and `bont4` benchmarks all have executions where the error of the end-to-end result decreases with an increasing AQM for the observed signals. The remaining executions all correlate with different measures. The error in the `cos` benchmark decreases with increasing speed, the error in the `kalman` benchmark decreases with an increasing `dqm`. The end-to-end error for the `vanderpol` benchmark is not well correlated with any single measure. For most benchmarks, the observed signal quality is most strongly correlated with the quality of the end-to-end result. However, no single quality measure or speed is strongly correlated with the end-to-end result across all benchmarks.

For all benchmarks except the `cos` and `spring` benchmarks, the `balanced` scaling objective value is much more strongly correlated with the end-to-end error than the execution speed or any one quality measure. This observation indicates that the inclusion of the AQMs, DQMs, and speed in the `balanced` execution formula enables it to better predict the result fidelity than any one quality measure. Note that for the `kalman` execution, only the `minimize_error` entry correlates strongly with the scaling objective value.

Conclusion: The `balanced` scaling objective function is an effective predictor of the end-to-end result for many kinds of applications. The `balanced` scaling objective does not adequately correlate with the end-to-end result quality though for some executions which target `maximize_fit` calibration strategy. A potential direction for future work would be the design of a better scaling objective for targeting the `maximize_fit` calibration strategy.

program	criteria	balanced	single	balanced	single
cos	% rmse	7.400e-02	6.880e-02		
cos	runtime	0.50 ms			
cos	power	0.19 mW			
cos	energy	0.10 μJ			
cosc	% rmse	3.332e+00	1.495e+00		
cosc	runtime	0.25 ms			
cosc	power	0.37 mW			
cosc	energy	0.09 μJ			
spring	% rmse	6.528e-01			
spring	runtime	0.94 ms	0.50 ms	3.375e+00	1.894e+00
spring	power	0.94 mW	0.91 mW	3.698e+00	2.685e+00
spring	energy	0.90 μJ	0.45 μJ	3.698e+00	1.894e+00
heatN4X2	% rmse	2.874e-02	2.646e-02		
heatN4X2	runtime	1.50 ms			
heatN4X2	power	0.72 mW			
heatN4X2	energy	1.09 μJ			
pid	% rmse	1.501e+00			
pid	runtime	4.79 ms	2.50 ms	2.124e+00	5.403e+01
pid	power	0.84 mW	0.82 mW	4.727e+00	7.036e+01
pid	energy	4.09 μJ	2.05 μJ	4.727e+00	5.403e+01
kalman	% rmse	2.853e-01	2.033e-02		
kalman	runtime	2.50 ms			
kalman	power	0.85 mW	0.84 mW	6.389e+00	2.903e-01
kalman	energy	2.12 μJ	2.10 μJ	6.389e+00	2.903e-01
smmrxn	% rmse	1.211e-03	1.125e-03		
smmrxn	runtime	0.50 ms			
smmrxn	power	0.52 mW			
smmrxn	energy	0.26 μJ			
gentog	% rmse	9.439e+00			
gentog	runtime	0.69 ms	0.50 ms	9.876e+01	1.627e+01
gentog	power	1.07 mW	1.05 mW	3.597e+01	1.627e+01
gentog	energy	0.76 μJ	0.53 μJ	1.163e+02	1.627e+01
bont4	% rmse	7.905e-05			
bont4	runtime	0.25 ms			
bont4	power	1.00 mW	0.97 mW	1.201e-04	1.310e-04
bont4	energy	0.25 μJ	0.24 μJ	1.201e-04	1.310e-04

Table 10.18: Summary of `single` performance executions which outperform `balanced` executions. Any benchmarks which do not obtain performance improvements with single executions are omitted.

program	criteria	scale factor	expression
cos	% rmse	mag(integ(0,3,2,0),x)	((-1)*P)
cos	runtime		
cos	power		
cos	energy		
cosc	% rmse	mag(integ(0,3,0,0),x)	V
cosc	runtime		
cosc	power		
cosc	energy		
spring	% rmse		
spring	runtime	mag(integ(0,0,1,0),z)	VB
spring	power	mag(mult(0,0,0,0),z)	((-1.00)*fPB)
spring	energy	mag(integ(0,0,1,0),z)	VB
heatN4X2	% rmse	mag(fanout(0,3,0,0),z0)	D1
heatN4X2	runtime		
heatN4X2	power		
heatN4X2	energy		
pid	% rmse		
pid	runtime	mag(fanout(0,3,2,1),z0)	ERR
pid	power	mag(integ(0,3,0,0),x)	((10*(-0.25))*SIG)
pid	energy	mag(fanout(0,3,2,1),z0)	ERR
kalman	% rmse	mag(mult(0,3,0,1),x)	(2.00*(SIG)+(((-1)*X)))
kalman	runtime		
kalman	power	mag(mult(0,3,3,0),z)	((-0.04)*SIG)
kalman	energy	mag(mult(0,3,3,0),z)	((-0.04)*SIG)
smmrxn	% rmse	mag(mult(0,3,0,0),z)	((10*2.00)*((20*0.40))+(((-1)*ES)))
smmrxn	runtime		
smmrxn	power		
smmrxn	energy		
gentog	% rmse		
gentog	runtime	mag(lut(0,3,2,0),z)	(0.05*(15.60*pow((1) +(max((2*(0.50*((0.50*(U)+(U))*UMOD))),0)),(-1))))
gentog	power	mag(lut(0,3,2,0),z)	(0.05*(15.60*pow((1) +(max((2*(0.50*((0.50*(U)+(U))*UMOD))),0)),(-1))))
gentog	energy	mag(lut(0,3,2,0),z)	(0.05*(15.60*pow((1) +(max((2*(0.50*((0.50*(U)+(U))*UMOD))),0)),(-1))))
bont4	% rmse		
bont4	runtime		
bont4	power	mag(fanout(0,3,2,0),z0)	bulkB
bont4	energy	mag(fanout(0,3,2,0),z0)	bulkB

Table 10.19: `single` scale objective summary for best performing `single` executions for the `cos`, `spring`, `kalman`, `gentog`, and `bont4` benhcmarks.

10.9 Alternate Scaling Objective Functions

Is there a better scaling objective function than the `balanced` *scaling objective function?*

I next explore the potential benefits of engineering an alternate scaling objective function. In this section, I investigate the viability of the `single` scaling objective function introduced in Section 10.8 as a potential alternate scaling objective function.

- **Quality, Power, Energy, and Runtime**: I first compare the performance of this scaling objective to the `balanced` scaling objective used by the compiler. This analysis contrasts the quality, power, energy, and runtime characteristics of the two scaling objectives. This analysis aims to identify if these single-signal executions identify a new and desirable point in the tradeoff space.
- **Scaling Objective Characteristics**: I next investigate the characteristics of these signals to identify if any trends which can be used to develop new objective functions.

10.9.1 Quality, Power, Energy, and Runtime

Figure 10.18 presents the `single` executions which produce higher quality results or consume less time,power, or energy than the `balanced` executions. Each row in the table presents one `single` execution which outperforms all the `balanced` executions on a particular metric. The execution is selected by identifying the subset of `single` executions which outperform the `balanced` executions a target metric. I then selected `single` execution with the lowest error from that subset. Columns 1 and 2 present the benchmark and the target metric, column 3 and 4 present the metric values for the `balanced` and `single` executions, and columns 5 and 6 present the % rmses for the `balanced` and `single` executions. For example, for the `runtime` entry of the `spring` benchmark, the compiler produces a `single` execution which completes in 0.50 ms and reports a 1.894 % rmse. This is faster than the fastest `balanced` execution, which takes 0.94 ms and reports a `3.375` % rmse.

- **Quality (`% rmse` rows)**: For the `cos`, `cosc`, `heatN4X2`, `kalman`, and `smmrxn` benchmarks, the `single` scale objective is able to identify higher quality executions than the `balanced` scale objective. The accuracy of the `cosc` and `kalman` benchmarks significantly improve by 2.23x and 14.03x respectively. The `cos`, `heatN4X2`, and `smmrxn`

all report minor accuracy improvements of 1.07x, 1.08x, and 1.07x respectively. For these benchmarks, there is likely a subset of signals within the ADP which have a disproportionate effect on the end-to-end result. For the `pend`, `spring`, `vanderpol`, `heatN4X2`, `pid`, `forced`, `gentog`, and `bont4` benchmarks, the `balanced` executions generally produce higher quality results than the `single` executions.

- **Runtime (`runtime` rows)**: For the `spring`, `pid`, and `gentog` benchmarks, the `single` scale objective identifies executions which run faster than the `balanced` executions. For the `spring` and `gentog` benchmarks, the `single` execution also reports a lower error relative to the fastest `balanced` execution. For the `pid` benchmark, the accuracy of the fast `single` is far worse than the accuracy of the fastest `balanced` execution. Therefore, for some benchmarks the `single` scale objective is able to identify faster executions which also accrue less error when compared to the fastest `balanced` execution.

  The compiler finds faster scaled ADPs with the `single` scale objective because one of the invocations directly maximizes the time scale factor for one of its execution. Note that this outcome is consistent with the execution speed utilization results presented in Section 10.5.3. In it, the `spring`, `pid`, and `gentog` benchmarks all failed to attain the maximum possible execution speed with the `balanced` scale objective.

- **Power (`power` rows)**: For the `spring`, `pid`, `kalman`, `gentog` and `bont4` benchmarks, the `single` scale objective identifies executions which consume marginally less power than the `balanced` executions. Of these benchmarks, the `spring`, `kalman`, and `gentog` benchmarks also report lower errors than the lowest power `balanced` execution. For the `pid` benchmark, the accuracy of the fast `single` is far worse than the accuracy of the lowest-power `balanced` execution. Therefore, the `single` scale objective is able to identify lower power executions for some benchmarks which also accrue less error than the lowest-power `balanced` execution.

  The `single` executions may attain a lower power consumption because the signals in the circuit are subject to a more relaxed minimum AQM and DQM requirement. This relaxation of the scaling problem frees the compiler to select lower power modes when scaling the circuit. In some cases, this relaxation also enables the compiler to produce higher fidelity results. The `single` executions for the `spring` and `kalman`

benchmarks both report lower errors than the `balanced` executions.

- **Energy (`energy` rows)**: For the `spring`, `kalman`, `pid`, `gentog`, and `bont4` benchmarks, the `single` scale objective identifies executions which consume less energy than the `balanced` executions. The `single` executions for the `spring`, `kalman`, and `gentog` benchmarks all report lower error than the lowest energy `balanced` executions. The `single` executions for the `pid` and `bont4` benchmarks both report a higher error. Note that the `bont4` benchmark reports a marginally higher error. Therefore, the `single` scale objective is able to identify lower energy executions for some benchmarks which also accrue less error than the lowest-energy `balanced` execution.

  For the `bont4` and `kalman` benchmarks, the lower energy consumption resulted from lower power consumption. For all other benchmarks, the lower energy consumption resulted from both lower power consumption and faster runtimes.

From this analysis, I can conclude that the `single` scaling objective enables the compiler to identify scaled ADPs that occupy promising new points in the tradeoff space. I next qualitatively investigate the characteristics of the `single` scaled ADPs reported in Table 10.18.

10.9.2 Analysis of Best-Performing Circuits

I next qualitatively analyze the scaling objective functions for the best-performing `single` executions for the benchmark applications presented in Table 10.18. Table 10.19 presents an an overview of the signals maximized by the `single` executions presented in Table 10.18. Columns 1 and 2 present the benchmark application and the performance metric the `single` execution outperforms the `balanced` executions on. Recall that each `single` execution maximizes a single signal or maximizes the execution speed. Columns 4 and 5 present the magnitude scale factor associated with the maximized signal and associated dynamical system expression implemented by that signal. If there is no `single` execution that outperforms the balanced executions for a particular metric, columns 4-6 are left blank. This table omits any benchmarks which do not report performant `single` executions.

For the `cos`, `cosc`, `spring`, `heatN4X2`, `pid`, and `bont4` benchmarks, the best-performing `single` executions all maximize signals which implement variables or negated variables.

There is likely a subset of dynamical system variables for these benchmark applications that disproportionately affect the result and must therefore be maximized. These high-impact dynamical system variables could perhaps be identified by analyzing the dynamical system directly before compilation.

For the `pid`, `kalman`, `smmrxn`, and `gentog` benchmarks, the best-performing `single` executions all maximize signals which calculate intermediate expressions in the ADP. For these benchmark applications, there is likely a more complex, specialized scaling objective exists which would strategically maximize the certain signals within the ADP. Identifying such signals would require some analysis which identifies which signals in the circuit have a disproportionate effect on the end-to-end result.

Therefore, from this analysis, I can conclude that there is likely no generally applicable heuristic for identifying the signals which have a disproportionate effect on the end-to-end result.

10.10 Realtime Case Studies

I next compile and execute the two real-time signal processing applications introduced in Section 4.13 of the thesis on the HCDCv2. Both signal processing applications accept an external analog signal as input, compute the signal continuously in real-time, and then emit the result. I provided the the external signal by routing the output of an external signal generator to the `cin` block at location `(1,3,3,0)` on the HCDCv2.

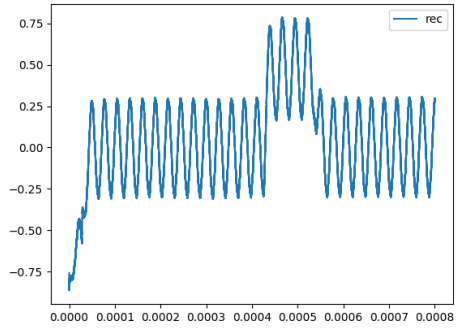


Figure 10-9: External input signal provided to bias shift detector.

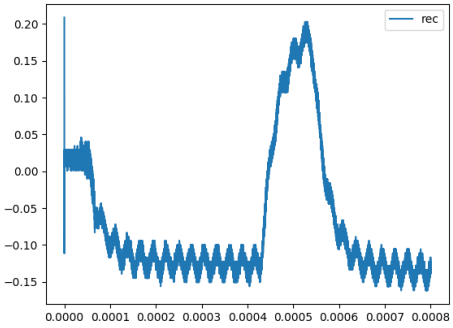


Figure 10-10: Output signal of bias shift detector.

10.10.1 Case Study A: Bias Shift Detector

The bias shift detector identifies shifts in the average value of an external signal. The output of the shift detector changes levels when a change in the average is detected. For this case study, I provide the HCDCv2 with an oscillating signal. This oscillating signal is centered around zero, has a frequency of 40 kHz, and has an amplitude of 0.25 V.

Figure 10-9 presents the analog signal provided to the device. As 4.5 milliseconds, the average of the oscillating signal shifts from zero to `0.5`. The average then shifts back down to zero after one millisecond. A simple thresholding operation would not adequately detect this shift in the average since the signal oscillates.

I compiled the bias shift detection dynamical system from Section 4.13.1 to the HCDCv2 and measured the externally accessible denoised signal produced by the application. The compiler scales the signal being emitted at the observation block `cout` by 0.4916x. The voltage of the measured signal is, therefore, 0.4916x the dynamical system observation variable. The time scale factor is `0.3016` which translates to a frequency of `0.3016*126` kHz, or 38 kHz – this is within the acceptable frequency range of 38-42 kHz for the benchmark application.

Figure 10-10 presents the output signal emitted by the bias shift detector after the recovery transform has been applied. The detector produces a signal with an amplitude of `-0.10` or a voltage of `49.16` mV when the average of the signal is zero. When the average of the external signal shifts to 0.5, the output of the bias detector also shifts to `0.15` or `7.374` mV. Note that the gain of the implemented denoiser is `0.5`x – this is higher than

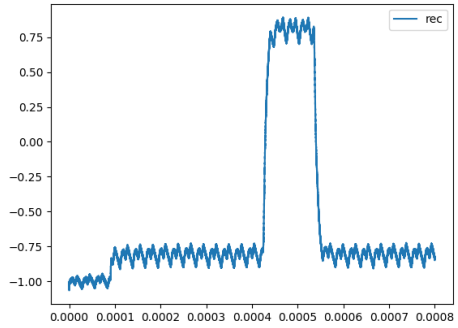


Figure 10-11: External input signal provided to denoiser.

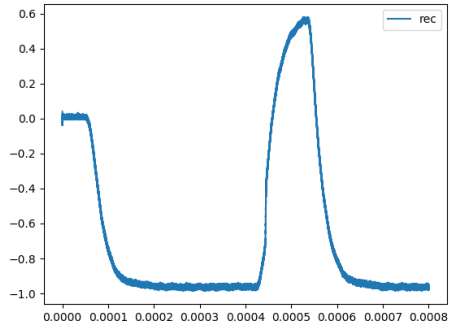


Figure 10-12: Output signal from denoiser.

the expected gain of `0.3`. This higher gain is not an issue for this use-case as a larger gain increases the dynamic range of the output. This larger than expected dynamic range enables thresholding circuits to more clearly delineate between different average values.

The signal produced by the bias detector can wake up a co-processor when a change in the sensed value occurs. The HCDCv2 output signal can be provided to a thresholding circuit that generates an interrupt when the voltage passes five millivolts. The main processor can then use this trigger to wake up the system.

10.10.2 Case Study B: Denoiser

The bias shift detector smoothes a noisy, externally provided signal and emits the denoised output. For this case study, I supply the HCDCv2 with a noisy pulse as input. The pulse has an amplitude of 1.5 V, lasts for one millisecond, and is offset from zero by -0.75 V.

Figure 10-11 presents the analog signal provided to the device. As 4 milliseconds, the signal shifts from `-0.75` to `0.75`. The average then shifts back down to `-0.75` after one millisecond.

I compiled the denoiser dynamical system from Section 4.13.2 to the HCDCv2 and measured the externally accessible denoised signal produced by the application. The compiler scales the signal being emitted at the observation block `cout` by 0.8948x. The voltage of the measured signal is, therefore, 0.8948x the dynamical system observation variable. The time scale factor is `0.3175` which translates to a frequency of `0.3175*126` kHz, or 40 kHz – this is within the acceptable frequency range of `38-42` kHz for the benchmark application.

Figure 10-10 presents the output signal emitted by the denoiser after the recovery transform has been applied to the measured signal. The detector produces a signal with an amplitude of `-1.0` or a voltage of `-0.895` V before the pulse is dispatched. While the pulse is active, the denoiser shifts to `0.6` or `0.537` V. Note that the leading edge of the pulse is not crisp in the denoised output. This is because the denoiser internally implements an average-tracking Kalman filter. This average tracking Kalman filter is configured to slowly adjust its internal state. It, therefore, takes some time to process the discontinuity at four milliseconds.

The denoised signal produced by the HCDCv2 can be provided to a DAC with a lower sampling rate and further processed on a digital machine. The HCDCv2 is an appealing platform for this case because it can implement real-time signal processing operations which can operate on higher frequency signals. These analog signal processing computations produce lower frequency signals as output which can easily be sampled and processed by a low-power digital device such as a microcontroller.

10.11 Conclusion

I built out a compilation toolchain that targets the HCDCv2 reconfigurable analog device and evaluated the compilation toolchain on twelve benchmark applications from the biology, physics, and controls domains. The lowest-error compiled benchmark applications execute in 0.25-1.92 ms, consume 0.10-5.09 μJ of energy ad 0.20-1.10 mW of power. The waveforms produced by the compiled circuits were indistinguishable from the original dynamical system trajectories.

Importance of Compiler Optimizations: I next investigated the importance of different compiler optimizations on the end-to-end result. I determined that the circuit scaling procedure is integral to producing executable ADPs for all benchmark applications. I also evaluated the importance of the mode modification feature implemented by the circuit scaling procedure. I confirmed that for 8 of the 12 benchmarks, the mode modification feature enabled the circuit scaling procedure to produce much higher quality results. I then evaluated the importance of the delta model compensation feature implemented by the circuit scaling procedure. For 11 of the 12 benchmarks, delta model compensation improves the quality of the produced result.

Performance of Co-Designed Calibration Strategy: I then investigated whether the co-designed calibration strategy produces better results than the traditional calibration strategy. The co-designed calibration strategy prioritizes eliminating uncorrectable unwanted behaviors present in the blocks. This strategy, therefore, allows for block behavior to deviate from the expected behavior, provided the compiler can handle these deviations in compilation. I found that for 10 of the 12 benchmark applications, the co-designed calibration strategy produces lower error results. For 8 of the 12 benchmark applications, the co-designed calibration strategy more consistently produces results of the same quality. From these results, I can conclude the co-designed strategy calibrates the device to behave more predictably and eliminates more unmitigateable unwanted behaviors than the traditional strategy.

Compilation Times: I then presented the compilation times for the benchmark applications. The circuit synthesis pass takes between 1.49 to 31.84 seconds to synthesize a circuit, and the circuit scaling pass takes between 0.13 to 6.13 seconds to scale a given circuit.

ADP Optimality: I then studied the optimality of the unscaled circuits. For all of the benchmarks, the compiled ADPs use the minimum number of assembly blocks. For 11 of the 12 benchmarks, the compiled ADPs use the minimum number of route blocks. For 10 of the 12 benchmarks, the compiled ADPs introduce additional multipliers into the computation when synthesizing the circuit. The compiler uses these multiplier blocks to compensate for constant values introduced by the analog blocks. The compiler uses the minimum number of all other compute blocks.

I then studied the optimality of the scaled circuits. For 9 of the 12 benchmarks, the compiler produces scaled circuits that execute at (at least) 95% of the maximum speed supported by the device. Generally speaking, the circuit scaling procedure cannot simultaneously maximize all signals and values in the ADP. However, the compiler can deliver a good dynamic range of a significant fraction of signals. Signals with a high dynamic range are less affected by noise and quantization error. For all of the benchmarks, the compiler produces at least one ADP where at least half of the signals use 50% of the available operating range. The compiler can also scale up the constant values so that they are less affected by quantization error. For all of the benchmarks, the compiler produces at least one ADP where at least half the value amplitudes are greater than `0.5` – the HCDCv2 supports digital values between -1 and 1.

I also investigated how effectively the scaled circuits minimized the `balanced` scaling objective function used by the compiler. The circuit scaling pass was generally able to identify multiple viable scaling transforms that all minimize the `balanced` scale objective function. I found that for all applications, the median objective function values of the scaled circuits were orders of magnitude better than the worst-case objective function value. For each benchmark, the spread of scale objective function values is also relatively small.

I then investigated the distribution of quality measures for each scaled circuit. I found that the quality measures varied substantially within each benchmark. The compiler, therefore, identifies multiple close-to-optimal scaling transforms which have different quality characteristics.

Unscaled ADPs: I then investigated why the unscaled ADPs produced by the compiler cannot be run on the HCDCv2. I found that all unscaled ADPs violate the operating range and frequency constraints imposed by the hardware. For eleven of the twelve benchmarks, all of the unscaled ADPs contain at least one data field value that falls outside the range of values supported by the hardware. These unscaled ADPs cannot even be written to the analog hardware.

Scaled ADP Complexity: I then investigated the complexity of the scale transforms produced by the compiler. The goal of this analysis was to identify if the scaling transforms were complex enough to warrant automated analysis. The benchmark applications contained between 14-90 magnitude scale factors (4-30 unique values), 0-6 injected values and one time scale factor. The scale factors took on a large range of values. The compiler, therefore, identifies complex scaling transforms comprised of multiple distinct scale factors.

ADP Characteristics and Result Quality: I then studied the relationship between various ADP characteristics and the end-to-end result quality. I discovered that the block instance selections have a significant effect on the quality of the end-to-end result. This observation is not currently factored into the compilation problem. I also was able to confirm that the `balanced` scaling objective value is predictive of the quality of the end-to-end result for 11 of the 12 benchmark applications. This finding indicates that the `balanced` scaling objective is a good heuristic for end-to-end result quality.

Alternate Scaling Objective Functions: I explored an alternative scaling objective function to the `balanced` scaling objective that maximizes a single signal or value. The goal of this analysis was to identify it is possible to engineer a scaling objective function that

improves on the `balanced` scale objective function. I was able to confirm that, for a subset of the benchmarks, this alternate scaling objective is able to unlock lower energy or higher performance executions with comparable or better error characteristics.

Realtime Case Studies: I presented two case studies in which the HCDCv2 is configured to compute on an external, continuously evolving analog signal. In both cases, the HCDCv2 is able to analyze the signal in real-time and produce satisfactory results.

Further Reading: Refer to Chapter 6 for a qualitative analysis of each of the best-performing scaled ADPs reported in this chapter.

Chapter 11

Conclusion

In recent years, analog computation has been experiencing a renaissance in the hardware community. Hardware designers have put forth a variety of modern day digitally programmable electrical analog devices which are capable of efficiently performing a variety of computations [51, 61, 54, 11, 105, 29, 128, 31, 140, 109, 15, 102]. These analog devices use standard CMOS processes and leverage transistor physics to perform computation. This line of research focuses on ultra-low power reconfigurable electrical analog devices which simulate dynamical system computations [61, 51, 128, 140]. Dynamical system-solving analog devices are attractive computational targets because they are low-power devices, have predictable performance characteristics, and can directly interface with analog sensors and actuators [51, 61].

In this thesis, I present a compiler that automatically programs a reconfigurable dynamical system-solving analog device to implement a target dynamical system. This is the first compiler to automatically target a reprogrammable analog device of this class. The presented compiler frees the end-user from reasoning about the low-level analog behaviors present in the hardware and automates the process of building the analog circuit that implements the desired computation. The compiler is able to reason about a variety of low-level physical behaviors, including analog noise, quantization error, process variation-induced behavioral variations, and signal and frequency range limitations.

The compiler configures the device so that the original dynamical system dynamics can be recovered from the circuit physics at runtime through the use of a compiler-derived recovery transform. The compiler transforms the target computation to respect the physical

limitations of the hardware and attenuate unwanted physical behaviors from the computation. Prior to this work, practitioners would have to manually derive and transform the programmed circuit to account for low-level behaviors while simultaneously ensuring the original dynamical system is recoverable at runtime. The compiler automates this arduous, error-prone programming process and decreases the barrier to adopting these platforms today.

This chapter reflects on the high-level design decisions made when pursuing this work and identifies, to what extent, the developed techniques can be applied to other hardware platforms. I conclude this chapter by discussing limitations and directions for future work.

11.1 Review

The work presented in this thesis demonstrates that it is possible to automatically program dynamical system-solving reconfigurable analog devices to implement dynamical system computations with acceptable accuracy. In this section, I explore the rationale behind the high-level decisions made when designing the compiler. This section overviews the following topics:

- **Circuit Scaling**: I provide rationale for why I chose to work with scaling transforms composed of constant factors and outline the benefits of the convex optimization problem formulation. I discuss the relationship between the sophistication of the circuit transform and the complexity of the optimization problem.
- **Calibration, Delta Models, and Software Compensation**: I discuss the key insights behind designing the cross-cutting compiler optimization that reasons about behavioral variations. I then reflect on how these insights can potentially applied to other hardware platforms.
- **Analog Device Specification Language**: I outline the design decisions I made when designing the analog device specification language. I provide a set of best practices, substantiated with personal experience, for creating analog hardware abstractions.
- **Circuit Synthesis**: I discuss the generality of the circuit synthesis pass and motivate the decisions behind the chosen circuit synthesis problem decomposition. I describe

the effect this decomposition has on the performance of the compiler and the resource efficiency of the produced circuit.

In this section, I evaluate the generality of the presented compilation techniques and discuss to what extent they can be reused in future work. I anticipate that many of the methods presented in this thesis are amenable to reuse and can be re-purposed to target other kinds of hardware platforms.

11.1.1 Circuit Scaling

The low-level physics of the device has a fundamental effect on the computation. Analog blocks within the hardware impose frequency, current, and voltage range limitations and have unique noise and error characteristics. Analog blocks are also subject to process variation-induced variations in behavior. These behavioral variations differ from fabricated device to fabricated device. All of these analog behaviors can adversely affect the mapped computation.

The compiler presented in this thesis automatically reasons about all of these low-level behaviors when mapping the dynamical system to the analog hardware. The compiler deploys a circuit scaling procedure that automatically transforms a circuit to respect the physical constraints of the device. The signals in the transformed circuit abide by the operating range and frequency limitations imposed by the device and have a large enough dynamic range to overcome the effects of noise and quantization error. The transformed circuit also compensates for behavioral deviations present on the device. The transformed circuit also preserves the original behavior of the dynamical system such that it can be recovered at runtime by applying an inverting transform.

Generality: I anticipate this automated scaling technique will be able to consider and, in some cases, compensate for a wide range of other physical phenomena. The circuit scaling procedure formulates the core problem of identifying a scaling transform as a geometric programming problem. A geometric programming problem is a convex optimization problem that supports non-linear constraints over positive, non-zero, real-valued variables. This formulation can currently capture operating range and bandwidth limitations, analog noise, a subset of process variation-induced behavioral variations, and digital quantization error. Though these constraints are specific to this class of reprogrammable analog devices, the

problem formulation and associated transform are general and can likely be applied more broadly.

Expressivity: The scaling transform formulation used in this work is both expressive and easy to efficiently derive from the circuit characteristics. A key challenge was identifying a sufficiently expressive transform that can be efficiently solved and effectively propagated through the circuit dynamics. I discuss the limitations I imposed on the transformation to ensure it can both be efficiently solved and effectively propagated below:

Positive, Non-Zero Scale Factors: The scaling transform supports scaling signals and data fields by positive, non-zero values. In my experience, excluding negative and zero scale factors does not appreciably reduce the power of the scaling transform for this target class of hardware. Scale factors should never equal zero, as it destroys the associated signal or value. Scaling signals by negative values may be helpful for analog devices which offer blocks that have asymmetrical operating ranges (e.g., `[-1,20]`). In my experience, this situation isn't common in practice because analog hardware platforms tend to offer symmetrical port operating ranges.

Extending the scaling transform to include negative values without changing the scaling problem formulation would render the scaling problem non-convex. The scaling problem could be formulated as a linear programming problem if both convexity and support for negative scale factors are desired. This formulation would not support the propagation of signals through non-linear block dynamics or support time scaling. These restrictions limit the compiler's ability to scale the circuit effectively.

Other Circuit Transforms: The scaling transform proposed in this work is more restrictive than other transformations, such as linear and polynomial transformations. In my experience, it is difficult to formulate the problem of deriving these more complex circuit transformations as convex optimization problems. These more complex transforms are also more difficult to propagate through blocks implementing non-linear functions. For these sorts of transformations, the compiler can less productively use the transform to reduce the effect of physical behaviors on the computation.

11.1.2 Calibration, Delta Models, and Software Compensation

In this thesis, I presented a cross-cutting compiler optimization that enables the compiler to target the specific device on hand more effectively. The key insight behind this compiler optimization is that the circuit scaling pass of compilation can compensate for unexpected signal gains in the calibrated blocks. This optimization requires the device calibration algorithm, runtime system, analog device specification, and compiler to work together to target the hardware effectively. I present a general framework for capturing and compensating for unexpected behaviors present in the hardware. I introduce the concept of a delta model, a hardware abstraction that captures the behavioral deviations present in a configured block instance. The compiler targets the delta models for the device on hand when scaling the circuit.

Process variation-induced behavioral deviations are prevalent in a wide range of different mixed-signal and analog ICs [48, 23, 24, 88]. I anticipate this approach can compensate for many other behaviors present in other mixed-signal hardware platforms. Specifically, I expect the general delta model abstraction to be broadly applicable across devices. Depending on the target hardware platform, the specific software compensation methods and device calibration and characterization routines may vary.

In this thesis, I present a co-designed calibration strategy that prioritizes eliminates unwanted behaviors that cannot be compensated for in compilation. I believe hardware designers can use the insights behind the design of this calibration strategy to design more effective device calibration algorithms for other mixed-signal platforms. Traditionally, device calibration algorithms aim to eliminate all unexpected behavior present in the hardware. However, if some of these unexpected behaviors can be corrected in software, the hardware designer is free to design calibration algorithms that prioritize eliminating more problematic unwanted behaviors. This insight may also facilitate the development of more resource-efficient designs. For example, the hardware designer could opt to use simpler calibration circuits or select more resource-efficient designs which are more sensitive to process variation-induced errors.

11.1.3 Analog Device Specification Language

A key challenge in developing the compiler was identifying the right abstractions for the analog hardware. In this thesis, I presented the analog device specification language (ADSL), a specification language for describing the programmable blocks and connections available on the analog device. The analog device specification language supported defining blocks with specialized programming interfaces and offered language constructs for describing low-level physical behaviors present in the blocks. I believe compiler writers can use the lessons learned in designing the ADSL to develop hardware abstractions for other reconfigurable mixed-signal hardware platforms. I summarize these lessons below:

Programming Interfaces: It is integral to expose a rich hardware programming interface to the compiler-writer and perform as few programming decisions in the device firmware and runtime as possible. The compiler has more domain information at its disposal and can apply more aggressive program transformations to the computation. Therefore, the compiler-writer is better positioned to handle these low-level programming decisions than the runtime system and firmware. Though the compiler itself works a rich hardware interface, the compiler mustn't leak these low-level hardware details to the programmer, as this would affect the usability of the hardware.

Rationale: In this research, I lowered the level of abstraction for the target hardware and provided my compiler with fine-grain control of how each block is programmed. This decision was a critical factor in the success of this research. Initially, the hardware programming interface for the HCDCv2 operated at a much higher level abstraction. This interface did not allow the end user to fully configure the block modes or write calibration configurations to the blocks. The device firmware instead made these programming decisions at runtime. By lowering the abstraction level, I was able to target the hardware much more effectively and obtain higher fidelity results on the target platform.

Abstractions for Low-level Physical Behaviors: A good analog hardware abstraction should capture all the behaviors that matter at a level of abstraction that is amenable to automated reasoning. In my experience, it is not necessary to fully model all of the low-level behaviors in great detail to productively target a piece of analog hardware.

Rationale: When I developed the ADSL, I decided how the specification language should model each of the low-level analog behaviors present in the hardware. For example, in the

ADSL, I represent analog noise as a standard deviation – I chose this representation over a more detailed abstraction that captures the distribution of the noise. In this case, the simpler noise abstraction was preferable because it was easier for the compiler to reason about automatically. Even with this simplified noise model, the compiler is capable of producing circuits that deliver high-fidelity results. I also navigated these sorts of expressivity tradeoffs when designing the delta model specifications.

Capturing Complicated Physical Behaviors: Typically, the goal is to directly model the analog behavior and then design software techniques that automatically reason about this behavior to target the device. It is not always feasible to directly model analog behavior, as some analog behaviors are challenging to capture and even more difficult to automatically reason about.

For these sorts of challenging behaviors, it is often productive to impose physical restrictions in the hardware abstraction that ensure that the target analog behavior has a negligible effect on the computation. These physical restrictions may reduce the performance or fidelity of the mapped computations on the device but make the hardware easier to target with automated techniques. The compiler-writer may also choose to impose a physical restriction to reduce the effect of certain behaviors and reduce the scope of the compilation problem.

Rationale: When designing the ADSL, I had to identify the right abstraction for encoding the effects of frequency-dependent block behavior on the computation. Note that analog blocks often experience reductions in signal gain when working with high-frequency signals. Because this behavior is non-trivial to reason about automatically, I chose to instead specify a maximum frequency limitation that ensures the frequency-dependent behavior of each block has a negligible effect on the computation. With this abstraction, the compiler can effectively target the hardware without reasoning about frequency-dependent behavior, provided the produced computation honors the frequency restrictions imposed by the analog hardware specification. Note that this frequency restriction causes the compiler to produce slower computations since it limits the speed of the dynamical system on the analog hardware.

Prioritization of Physical Behaviors: It is important to prioritize accurately modeling the analog behaviors that have the largest effect on the computation. The compiler-writer should identify this prioritization by studying the characterization information collected from a fabricated device whenever possible.

Rationale: Early versions of the ADSL focused on modeling analog noise and did not focus on process-variation induced behavioral variations. Once I started working with the device, it became evident that static error introduced during fabrication had a sizeable effect on the computation even after the device was calibrated. This observation inspired the development of the delta model abstraction. In the final hardware abstraction, I modeled the behavioral variations for the target device on hand in great detail and opted to model analog noise as a fixed standard deviation. In short, I prioritized accurately modeling behavioral variations over analog noise because the static error had a more sizeable effect on the computation.

11.1.4 Circuit Synthesis

For dynamical system-solving reconfigurable analog devices, there is no universally agreed-upon collection of analog building blocks. Instead, this class of analog devices offers highly specialized reconfigurable analog blocks that may be configured to implement a variety of computations. These blocks implement anything from simple functions to differential equations. Some of these blocks may be special-use blocks that do not typically compute on signals. Examples of special use blocks include blocks that copy analog currents and blocks that route signals through the device.

The compiler's circuit synthesis procedure efficiently and automatically constructs analog circuits from non-standard computational and special-use blocks. The compiler employs a multi-stage, algebraic rewrite-based circuit synthesis procedure to map the dynamical system to the analog hardware.

Generality: The circuit synthesis procedure is a multi-stage procedure that handles non-standard computational blocks, special-use copier blocks, and special-use routing blocks in dedicated compilation passes. This compilation approach works best with current-mode analog devices containing special-use and parametric blocks. The specific synthesis techniques address specific architectural features that are likely to be prevalent in multiple analog devices. I anticipate compiler writers can use the compilation techniques employed in this thesis in conjunction with new approaches to design new compilers that efficiently target other reconfigurable mixed-signal architectures.

Resource Efficiency: The compiler deploys specialized circuit synthesis routines that intelligently introduce special-use blocks into the circuit. This decomposition enables the

compiler to produce resource-efficient, physically mappable circuits – a circuit is physically mappable if all of the blocks and connections can be mapped to physical block locations and physical interconnects on the hardware. Consider a more straightforward approach where the compiler does not distinguish between special-use and computational blocks during circuit synthesis. With this simpler approach, the compiler would introduce special-use blocks into the circuit unnecessarily or inappropriately since many of these blocks implement the equality relation and can therefore be inserted almost anywhere. The circuits produced by such an approach may be suboptimal and use extraneous copying and routing blocks or may not be physically mappable on the hardware.

I anticipate this circuit synthesis problem decomposition could be reused to produce efficient circuits for other reconfigurable mixed-signal hardware platforms that offer special-use blocks. This decomposition was architected with the insight that the compiler should deploy specialized algorithms that efficiently introduce special-use blocks into the synthesized circuit. Compiler designers may find this insight helpful when architecting new synthesis problem decompositions.

Performance: The architecture of the circuit synthesis procedure enables the compiler to produce circuits efficiently. Many of the employed compiler optimizations aim to reduce the size of the problems provided to the circuit synthesis subroutines – this is important as some of the synthesis subroutines scale poorly with increasing problem size. For example, the compiler uses the computationally expensive search-based circuit fragment synthesis algorithm to produce circuit fragments for each dynamical system relation and then assembles these fragments into a circuit with a lightweight assembly algorithm. With this decomposition, the performance of the circuit fragment synthesis algorithm depends on the complexity of the target algebraic relation rather than the dynamical system size. The circuit fragment synthesis invocations can also be parallelized with this decomposition. The drawback to using this decomposition is that it restricts the kinds of connections that can be made between circuit fragments and therefore limits the types of circuits that the compiler can generate. For example, with this compilation approach, the compiler cannot automatically reuse part of one circuit fragment to compute a subexpression in another circuit fragment.

In this work, I explore one possible compiler architecture that efficiently generates circuits for a broad range of applications for the target class of hardware. This compilation approach may not work well for all hardware platforms. For example, some hardware plat-

forms may require circuit patterns that cannot be produced with this compiler architecture. For this reason, I anticipate the overall compilation strategy may differ depending on the characteristics of the target device.

11.2 Limitations

I next discuss the limitations of the work presented in this thesis. I discuss the expressivity of the dynamical system and analog device specification languages and the limitations imposed by the circuit synthesis and circuit scaling compilation passes.

11.2.1 Expressivity of Dynamical System Specification Language

The dynamical system specification language presented in this work focuses on first-order explicit ordinary differential equations. This language does not offer constructs for defining implicit ordinary differential equations, time-delayed differential equations, partial differential equations, or stochastic differential equations.

Implementing these other classes of differential equations in analog requires the development of new analog functional units. For example, time-delayed differential equations, stochastic differential equations and require hardware functional units that implement time delays and statistical distributions. Partial differential equations require hardware units that take the derivative of a signal with respect to another signal. I believe that it is not productive to expand the expressivity of the specification language to include these classes of dynamical systems until these hardware units are readily available in mixed-signal systems.

Other classes of dynamical systems require the development of new software techniques. For example, including support for implicit differential equations would significantly complicate the circuit synthesis procedure since the compiler can no longer leverage the first-order notation to synthesize the circuit fragments implementing individual variables independently. While these dynamical systems are presently outside the scope of this work, they can eventually be targeted by the compiler with future work.

The dynamical system specification language supports a subset of explicit ordinary differential equations. More specifically, the specification language does not support functions

that explicitly reference time or variables that take on discrete values. These mathematical operators are excluded from the specification language as they are non-trivial implement in analog and likely require specialized handling. For example, modeling time in analog is challenging because time increases linearly in an unbounded fashion. These operations likely require specialized hardware mapping procedures which are outside of the scope of this work. Note that these custom mapping procedures may not be necessary if hardware designers draft new hardware units which support these operators.

In summary, the limitations of dynamical system-solving mixed-signal devices primarily inform the expressivity limitations imposed in the dynamical system specification language. In some cases, I also limit the expressivity of the dynamical system specification language to reduce the scope of the compilation problem. These restrictions on language expressivity can be lifted as more compilation techniques are developed.

11.2.2 Expressivity of Analog Device Specification Language

The analog device specification language focuses solely on continuous-time digital and analog blocks. The specification language does not support the specification of dynamic stateful digital components, such as flip-flops and memories, or clocked digital blocks, such as digital clocks. The analog device specification language also does not support blocks that implement more sophisticated mathematical operators such as probability distributions, partial derivatives, and time-delayed variables. These blocks were not prevalent in the hardware I targeted and were therefore not included in the language.

The analog device specification language offers language constructs for defining the operating range and frequency limitations, behavioral variations, and analog noise present in each block. These language constructs do not model these analog behaviors in a high degree of detail. For example, the analog noise annotations specify the analog noise as a standard deviation rather than a distribution. I choose to use this simplified view of the low-level analog behaviors because it is more amenable to static analysis. Note that the compiler can produce circuits that implement the desired computation at high fidelity with this simplified abstraction of the analog behaviors. In the future, it might be productive to provide a more detailed specification of the analog behaviors for hardware emulation purposes.

11.2.3 Compiler Limitations

Presently, the circuit synthesis pass is not guaranteed to find an analog circuit that implements the dynamical system if one exists. The fragment synthesis step of circuit synthesis does not identify all possible algebraic unifications for a given block when searching through circuit fragments. In practice, the circuit synthesis pass works well and can identify circuits for all of the explored dynamical system applications. However, I anticipate the circuit synthesis pass may fail to produce circuits for applications that require highly non-trivial algebraic rewrites. These sorts of rewrites are likely unsupported by the algebraic unification routine.

Presently, the circuit scaling pass does not guarantee that produced scaled circuit will execute the dynamical system with acceptable error. Instead, the scaling procedure introduces constraints that ensure the signals and values in the circuit overcome the noise floor. These constraints involve the analog and digital quality measures of the circuit. These measures serve as proxies for the signal-to-noise ratio of the analog and digital signals in the hardware. This approach produces scaled circuits that work well in practice but cannot guarantee that all produced circuits would execute the target computation with acceptable error.

11.3 Future Directions

I next discuss potential future directions. I first discuss potential future compiler optimizations that can be investigated immediately. The core insights behind these compiler optimizations are substantiated by the findings presented in Chapter 10. I then present some long-term research directions inspired by the findings in this work.

11.3.1 Compiler Optimizations

The analyses presented in this thesis lay the groundwork for several potential future compiler optimizations and analyses. These future extensions can potentially enable the compiler to produce analog circuits that yield higher fidelity results.

Presently, the compiler treats all block instances of a given block type as equally good during the place and route stage of compilation. In Chapter 10, I identified that the block

instance selections made during the place+route procedure actually have a profound impact on the quality of the end-to-end result. Therefore, one potential future research direction would involve designing new place+route procedures that consider block fidelity when assigning block placements. This optimization would enable the compiler to produce scaled ADPs that strategically use accurate blocks in parts of the circuit that require greater accuracy.

Because each device would likely have a limited number of high-fidelity block instances, it would also be productive to develop circuit analyses that identify which signals in a given circuit are sensitive to error. The compiler could then map the blocks which work with these error-sensitive signals to high fidelity block instances on the device. Such an analysis could also be used by the circuit scaling procedure to infer fine-grain signal-specific quality constraints for the scaled circuits.

Presently, the compiler works with a user-defined set of minimum quality bounds and scales all benchmarks with the `balanced` scale objective. In Chapter 10, I determined that for some benchmarks, there exist alternate scaling objectives that deliver better results than the `balanced` scaling objective. In the presented analysis, I generate a collection of alternate scaling objectives that are all subject to a set of loose minimum quality restrictions. I believe it would be productive to develop a circuit analysis that identifies the best scaling objective function for a given circuit and identifies a set of minimum quality bounds for that circuit. Such an analysis would enable the compiler to produce scaled ADPs that deliver higher fidelity results or deliver better performance and energy efficiency without sacrificing signal fidelity.

11.3.2 Mixed-Signal Computing Paradigms

One potential future research direction involves identifying programming paradigms for hybrid computations which involve both analog and digital hardware. This hybrid computational model takes advantage of the energy efficiency of dynamical system-solving analog substrates and the flexibility of digital logic. For example, feature detection applications may perform feature identification in analog and then switch to a digital computation when a feature is detected [78, 14]. For applications such as global constrained optimization, it would be productive to solve the constrained local optimization sub-problems in analog and

track the global optimization state with digital logic.

In this thesis, I map the entire dynamical systems computation onto the target analog device. To extend this work to support hybrid systems, I would embed the dynamical system specification language into a more conventional programming language for digital systems. To pursue this line of work, I would need to design programming interfaces for mediating between digital and analog parts of the program and identify to what extent the programmer can modify the analog computation at runtime. For example, with global optimization, the surrounding digital program might dynamically adjust the search space and initial guess of the local search depending on the results returned by the analog hardware. These modifications would affect the initial condition and interval bounds for the state variables in the computation. The scaling transform associated with the analog computation would need to be dynamically adjusted at runtime to account for interval ranges and data field values to support such a computation. This retargeting operation would need to incur minimal overhead to maximize energy savings.

11.3.3 Automated Design-Space Exploration

This thesis presents a compiler that targets a fabricated analog device with a fixed design. The target analog device imposes a variety of physical constraints which affect the efficacy of the compiler. These physical constraints cannot be modified post-fabrication. Relevant physical constraints include operating range and frequency limitations and limitations on the number of block instances and connections. The hardware designer made these high-level design decisions before device fabrication. The hardware designer may conservatively design the hardware to support a broad range of applications. For example, the hardware designer may opt to support signals with large dynamic ranges or offer an excess of block instances and connections. These decisions may unnecessarily increase the device area and power consumption.

One potential research direction involves leveraging the compilation techniques presented in this thesis to facilitate design space exploration. This line of research would enable hardware designers to make informed design decisions when designing reconfigurable analog hardware platforms. To pursue this line of work, I would extend the hardware specification to support parametric hardware designs with unknown design parameters. This research

direction aims to identify resource-efficient candidate parametrizations of the hardware specifications that effectively execute a set of representative dynamical systems. A key challenge would be designing a set of analyses that identify the hardware specification's optimal design parameters.

Novel Hardware Abstractions: One potential future research direction involves working closely with hardware designers to create new hardware abstractions that take into account the capabilities of state-of-the-art software techniques. These abstractions may enable designers to produce simpler, more resource-efficient, and more performant hardware designs that better fit the needs of the target domain. These improved hardware abstractions may, for example, allow for the hardware to exhibit unexpected behavioral deviations, provided these deviations can be compensated for in software.

Consider analog hardware platforms which leverage the physical behavior of materials to perform computation. For these devices, variations introduced by the fabrication process typically translate to deviations in behavior in the computational units. Designers, therefore, incorporate compensating calibration circuits into their design to correct behavioral deviations after fabrication. These circuits are digitally configured through an objective-driven process called calibration. I propose developing and implementing a general specification language that enables compiler writers to describe acceptable behavioral deviations – that is, behaviors that can be mitigated with software techniques. This specification can then be analyzed to derive the device's calibration algorithms and characterization and model inference procedures.

11.4 Concluding Thoughts

This thesis investigates compilation techniques for ultra-low power reconfigurable analog computing platforms which solve dynamical systems [61, 51, 128, 140]. In this work, I present a compiler that frees the end user from having to reason about the low-level analog limitations and behaviors present in the hardware and automates the process of deriving the circuit for the desired computation. This work lowers the barrier of entry for programming this class of devices and renders these devices more accessible to end users.

This work focuses on mixed-signal reconfigurable hardware designed to solve dynamical systems. There has recently been a proliferation of mixed-signal platforms and physical

substrates that perform computation [120, 56, 54, 11, 108, 85, 105, 29, 128, 31, 140, 132, 51, 61, 141, 15, 109, 102, 62, 87, 89, 38]. These specialized computing platforms are becoming pervasive and crucial for satisfying the computational needs of different domains. In some cases, these novel computing platforms enable the efficient computation of entire classes of computations. Researchers have developed such platforms for a variety of different domains, including machine learning, quantum computing, signal processing, robotics, and biology.

These computing platforms typically leverage certain analog behaviors to perform computation efficiently. Because there is rarely a perfect mapping between the problem domain and the target analog substrate, these platforms also likely exhibit analog behaviors for which there is no analog in the problem domain. These unwanted analog behaviors must be managed or mitigated in hardware or software for the device to reliably and faithfully perform computations.

I believe these nascent computing platforms would benefit greatly from the development of software techniques that automatically amplify desired analog behaviors and mitigate unwanted analog behaviors. Software-based techniques offer several advantages over hardware mitigation techniques. First, software techniques enable the hardware to execute computations at higher fidelity without introducing area, power, or performance overheads into hardware design. Second, software techniques can automate labor-intensive programming procedures and enable hardware researchers to more systematically explore the capabilities of their hardware. Third, the software development cycle is significantly faster than the hardware design cycle and is typically less capital intensive. As a result, researchers can rapidly develop multiple software-based techniques in a relatively short span of time for a target class of hardware. Software-based techniques can also aggressively optimize the computation since they operate at a higher level of abstraction and have more domain information at their disposal. These capabilities together make software-based mitigation approaches an attractive and promising direction for future research.

I anticipate the lessons learned targeting dynamical system-solving analog devices can inform and inspire future software techniques for other mixed-signal and analog computing substrates. While the specific set of analog behaviors to consider may vary depending on the computational platform at hand, the approach to abstracting and statically reasoning about analog behavior can be broadly applied to multiple devices. The development of novel physics-aware software techniques, such as those presented in this thesis, promises to unlock

the potential of these non-traditional computing substrates and enables the development of a large range of non-standard computing platforms. With this potential unlocked, these non-standard computing platforms may then deliver significant performance, area, and energy benefits. I anticipate these novel physics-aware software techniques will also promote the exploration of new, non-standard, and transformative points in the hardware design space. These hardware technologies have the potential to significantly increase our computational capabilities, reduce the design, fabrication, and disposal costs associated with hardware production, and enable groundbreaking advances in our ability to compute, sense, and interact with physical world.

Bibliography

[1] *Handbook of Tableau Methods*. Springer, 1999.

[2] Sara Achour and Martin Rinard. Time dilation and contraction for programmable analog devices with jaunt. In *ACM SIGPLAN Notices*, volume 53, pages 229–242. ACM, 2018.

[3] Sara Achour and Martin Rinard. Noise-aware dynamical system compilation for analog devices with legno. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 149–166, 2020.

[4] Sara Achour, Rahul Sarpeshkar, and Martin C Rinard. Configuration synthesis for programmable analog devices with arco. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–193. ACM, 2016.

[5] Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.

[6] Thomas Carey Alan Carlson, George Hannauer and Peter J. Holsberg. *Handbook of Analog Computation, Second Edition*, volume 37. Electronic Associates, Inc, 1967.

[7] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

[8] Ludwig Arnold. Stochastic differential equations. *New York*, 1974.

[9] Karl Johan Åström and Tore Hägglund. *PID controllers: theory, design, and tuning*, volume 2. Instrument society of America Research Triangle Park, NC, 1995.

[10] Kendall Atkinson, Weimin Han, and David E Stewart. *Numerical solution of ordinary differential equations*, volume 108. John Wiley & Sons, 2011.

[11] Arindam Basu, Stephen Brink, Craig Schlottmann, Shubha Ramakrishnan, Csaba Petre, Scott Koziol, Faik Baskaya, Christopher M Twigg, and Paul Hasler. A floating-gate-based field-programmable analog array. *IEEE Journal of Solid-State Circuits*, 45(9):1781–1794, 2010.

[12] Randall D Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995.

[13] Alfredo Bellen and Marino Zennaro. *Numerical methods for delay differential equations*. Oxford university press, 2013.

[14] Jacob Benesty, Israel Cohen, and Jingdong Chen. *Array Beamforming with Linear Difference Equations*, volume 20. Springer Nature, 2021.

[15] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Shobhit Choudhary, Anand R Chandrasekaran, Jean-Marie Bussat, Rodrigo Alvarez-Icaza, John V Arthur, Paul Merolla, Kwabena Boahen, et al. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[16] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[17] David F Bizup and Donald E Brown. The over extended kalman filter- don't use it! In *Proceedings of the Sixth International Conference of Information Fusion*, volume 1, pages 40–46. Citeseer, 2003.

[18] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A tutorial on geometric programming. *Optimization and engineering*, 8(1):67, 2007.

[19] R.W. Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its Applications*, 146:79–91, 1991.

[20] AA Brown and MC Bartholomew-Biggs. Ode versus sqp methods for constrained optimization. *Journal of optimization theory and applications*, 62(3):371–386, 1989.

[21] Donald G Buerk. Can we model nitric oxide biotransport? a survey of mathematical models for a simple diatomic molecule with surprisingly complex biological activities. *Annual review of biomedical engineering*, 3(1):109–143, 2001.

[22] John Charles Butcher. A history of runge-kutta methods. *Applied numerical mathematics*, 20(3):247–260, 1996.

[23] Chih-Cheng Chang, Pin-Chun Chen, Teyuh Chou, I-Ting Wang, Boris Hudec, Che-Chia Chang, Chia-Ming Tsai, Tian-Sheuan Chang, and Tuo-Hung Hou. Mitigating asymmetric nonlinear weight update effects in hardware neural network based on analog resistive synapse. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):116–124, 2017.

[24] Chih-Cheng Chang, Jen-Chieh Liu, Yu-Lin Shen, Teyuh Chou, Pin-Chun Chen, I-Ting Wang, Chih-Chun Su, Ming-Hong Wu, Boris Hudec, Che-Chia Chang, et al. Challenges and opportunities toward online training acceleration using rram-based hardware neural network. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 11–6. IEEE, 2017.

[25] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

[26] S Alexander Chin and Jason H Anderson. An architecture-agnostic integer linear programming approach to cgra mapping. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[27] Krishna Choudhary and Atul Narang. Analytical expressions and physics for single-cell mrna distributions of the lac operon of e. coli. *Biophysical journal*, 117(3):572–586, 2019.

[28] George F Corliss. Survey of interval algorithms for ordinary differential equations. *Applied Mathematics and Computation*, 31:112–120, 1989.

[29] G.E.R. Cowan, R.C. Melville, and Y. Tsividis. A VLSI analog computer/digital computer accelerator. *Solid-State Circuits, IEEE Journal of*, 41(1):42–53, Jan 2006.

[30] Nick Csicsery and Ricky O'Laughlin. A mathematical model of a synthetically constructed genetic toggle switch. *Mathematical method in Bioengineering Report*, 2013.

[31] Ramiz Daniel, Sung Sik Woo, Lorenzo Turicchia, and Rahul Sarpeshkar. Analog transistor models of bacterial genetic circuits. In *Biomedical Circuits and Systems Conference (BioCAS), 2011 IEEE*, pages 333–336. IEEE, 2011.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[33] Richard C.. Dorf and Robert H Bishop. *Modern control systems*. Pearson Prentice Hall, 2008.

[34] J.L. Douce and H. Wilson. The automatic synthesis of control systems with constraints. *Mathematics and Computers in Simulation*, 7(1):18 – 22, 1965.

[35] Helmut Emmelmann, F-W Schröer, and Rudolf Landwehr. Beg: a generator for efficient back ends. In *ACM Sigplan Notices*, volume 24, pages 227–237. ACM, 1989.

[36] Kamil Erguler and Michael PH Stumpf. Practical limits for reverse engineering of dynamical systems: a statistical analysis of sensitivity and parameter inferability in systems biology models. *Molecular BioSystems*, 7(5):1593–1602, 2011.

[37] Jason Ernst and Manolis Kellis. ChromHMM: automating chromatin-state discovery and characterization. *Nature Methods*, 9(3):215–6, mar 2012.

[38] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[39] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[40] DJ Evans. Hybrid computation. ga bekey and wa karplus. john wiley, 1968. 464 pp. illustrated. 125s. *The Aeronautical Journal*, 73(708):1052–1052, 1969.

[41] Edward H Flach and Santiago Schnell. Use and abuse of the quasi-steady-state approximation. *IEE Proceedings-Systems Biology*, 153(4):187–191, 2006.

[42] MA Franklin and JC Strauss. Automated programming of analog hybrid computers: —a review. *Simulation*, 18(1):11–19, 1972.

[43] Mark A Franklin and Jon C Strauss. A hybird computer programming system. In *Proceedings of the November 18-20, 1969, fall joint computer conference*, pages 275–285, 1969.

[44] Christopher W Fraser, David R Hanson, and Todd A Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):213–226, 1992.

[45] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[46] Timothy S Gardner, Charles R Cantor, and James J Collins. Construction of a genetic toggle switch in escherichia coli. *Nature*, 403(6767):339–342, 2000.

[47] Eva M. Golos, Hongjian Fang, and Robert D. van der Hilst. Variations in seismic wave speed and vp/vs ratio in the north american lithosphere. *Journal of Geophysical Research: Solid Earth*, 125(12):e2020JB020574, 2020.

[48] Sujan Kumar Gonugondla, Mingu Kang, and Naresh Shanbhag. A 42pj/decision 3.12 tops/w robust in-memory machine learning classifier with on-chip training. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 490–492. IEEE, 2018.

[49] Clement Green, Hervé D'Hoop, and André Debroux. Apache-a breakthrough in analog computing. *IRE Transactions on Electronic Computers*, (5):699–706, 1962.

[50] Ning Guo. *Investigation of Energy-Efficient Hybrid Analog/Digital Approximate Computation in Continuous Time*. PhD thesis, Columbia University, 2017.

[51] Ning Guo, Yipeng Huang, Tao Mai, Sharvil Patil, Chi Cao, Mingoo Seok, Simha Sethumadhavan, and Yannis Tsividis. Energy-efficient hybrid analog/digital approximate computation in continuous time. *IEEE Journal of Solid-State Circuits*, 51(7):1514–1524, 2016.

[52] Carroll Ray Hall and SJ Kahne. An improved method for analog computer scaling. *Mathematics and Computers in Simulation*, 12(1):27–32, 1970.

[53] Carroll Ray Hall and Stephen J Kahne. Automated scaling for hybrid computers. *IEEE Transactions on Computers*, 100(5):416–423, 1969.

[54] Tyson S Hall, Christopher M Twigg, Jordan D Gray, Paul Hasler, and David V Anderson. Large-scale field-programmable analog arrays for analog signal processing. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 52(11):2298–2307, 2005.

[55] Ensign John H Hanna and Ensign Harold E Millan Jr. Automated analog programming in hybrid systems: A method of making the analog computer accessible to all engineers. *Naval Engineers Journal*, 78(5):895–899, 1966.

[56] Jennifer Hasler. Opportunities in physical computing driven by analog realization. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2016.

[57] Timothy Hickey, Qun Ju, and Maarten H Van Emden. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001.

[58] Desmond J Higham and Lloyd N Trefethen. Stiffness of odes. *BIT Numerical Mathematics*, 33(2):285–303, 1993.

[59] Nicholas J Higham, D Steven Mackey, Françoise Tisseur, and Seamus D Garvey. Scaling, sensitivity and stability in the numerical solution of quadratic eigenvalue problems. *International journal for numerical methods in engineering*, 73(3):344–360, 2008.

[60] Fritz Horn and Roy Jackson. General mass action kinetics. *Archive for rational mechanics and analysis*, 47(2):81–116, 1972.

[61] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsividis, Kyle Mandli, and Simha Sethumadhavan. Hybrid analog-digital solution of nonlinear partial differential equations. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 665–678. IEEE, 2017.

[62] Yipeng Huang, Ning Guo, Mingoo Seok, Yannis Tsividis, and Simha Sethumadhavan. Analog computing in a modern context: A linear algebra accelerator case study. *IEEE Micro*, 37(3):30–38, 2017.

[63] Electronic Associates Inc. *Pace EAI 231R Manual*. 1961.

[64] Electronic Associates Inc. *EAI 680 Reference Handbook*. 1966.

[65] Electronic Associates Inc. *EAI 680 scientific computing system: Brochurean economical, high-performance, hybrid computer*. 1966.

[66] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis: With Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer London, 2012.

[67] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. Interval analysis. In *Applied interval analysis*, pages 11–43. Springer, 2001.

[68] Rajeev Joshi, Greg Nelson, and Keith Randall. *Denali: a goal-directed superoptimizer*, volume 37. ACM, 2002.

[69] Dion Khodagholy, Jennifer N Gelinas, Thomas Thesen, Werner Doyle, Orrin Devinsky, George G Malliaras, and György Buzsáki. Neurogrid: recording action potentials from the surface of the brain. *Nature neuroscience*, 18(2):310–315, 2015.

[70] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311, 2018.

[71] L.V. Kolev. *Interval Methods for Circuit Analysis*. Advanced series on circuits and systems. World Scientific, 1993.

[72] Kenneth S Krane. *Modern physics*. John Wiley & Sons, 2019.

[73] Yang Kuang. *Delay differential equations*. University of California Press, 2012.

[74] E Lalonde, A S Ishkanian, J Sykes, M Fraser, H Ross-Adams, N Erho, M J Dunning, S Halim, A D Lamb, N C Moon, G Zafarana, A Y Warren, X Meng, J Thoms, M R Grzadkowski, A Berlin, C L Have, V R Ramnarine, C Q Yao, C A Malloff, L L Lam, H Xie, N J Harding, D Y Mak, K C Chu, L C Chong, D H Sendorek, C P'ng, C C Collins, J A Squire, I Jurisica, C Cooper, R Eeles, M Pintilie, A Dal Pra, E Davicioni, W L Lam, M Milosevic, D E Neal, T van der Kwast, P C Boutros, and R G Bristow. Tumour genomic and microenvironmental heterogeneity for integrated prediction of 5-year biochemical recurrence of prostate cancer: a retrospective cohort study. *Lancet Oncol*, 15(13):1521–1532, 2014.

[75] J Paul Landauer. Program-generation system for modern hybrid computers. *Simulation*, 26(6):169–176, 1976.

[76] Frank J Lebeda, Michael Adler, Keith Erickson, and Yaroslav Chushak. Onset dynamics of type a botulinum neurotoxin-induced paralysis. *Journal of pharmacokinetics and pharmacodynamics*, 35(3):251, 2008.

[77] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

[78] Frank L Lewis, Lihua Xie, and Dan Popa. *Optimal and robust estimation: with an introduction to stochastic control theory*. CRC press, 2017.

[79] Jifu Liang, Nilan Udayanga, Arjuna Madanayake, S. I. Hariharan, and Soumyajit Mandal. An offset-cancelling discrete-time analog computer for solving 1-d wave equations. *IEEE Journal of Solid-State Circuits*, pages 1–1, 2021.

[80] FMS Lima and P Arun. An accurate formula for the period of a simple pendulum oscillating beyond the small angle regime. *American Journal of Physics*, 74(10):892–895, 2006.

[81] Xing Lin, Yair Rivenson, Nezih T Yardimci, Muhammed Veli, Yi Luo, Mona Jarrahi, and Aydogan Ozcan. All-optical machine learning using diffractive deep neural networks. *Science*, 361(6406):1004–1008, 2018.

[82] Yonathan Malachi, Zohar Manna, and Richard Waldinger. Tablog: The deductive-tableau programming language. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 323–330, 1984.

[83] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121, 1980.

[84] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *Software Engineering, IEEE Transactions on*, 18(8):674–704, 1992.

[85] Peter L McMahon, Alireza Marandi, Yoshitaka Haribara, Ryan Hamerly, Carsten Langrock, Shuhei Tamate, Takahiro Inagaki, Hiroki Takesue, Shoko Utsunomiya, Kazuyuki Aihara, et al. A fully programmable 100-spin coherent ising machine with all-to-all connections. *Science*, 354(6312):614–617, 2016.

[86] J Kyle Medley, Jonathan Teo, Sung Sik Woo, Joseph Hellerstein, Rahul Sarpeshkar, and Herbert M Sauro. A compiler for biological networks on silicon chips. *PLoS computational biology*, 16(9):e1008063, 2020.

[87] Botond Molnár, Ferenc Molnár, Melinda Varga, Zoltán Toroczkai, and Mária Ercsey-Ravasz. A continuous-time maxsat solver with high analog performance. *Nature communications*, 9(1):4864, 2018.

[88] Boris Murmann. Mixed-signal computing for deep neural network inference. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(1):3–13, 2020.

[89] Alexander Neckar, Sam Fok, Ben V Benjamin, Terrence C Stewart, Nick N Oza, Aaron R Voelker, Chris Eliasmith, Rajit Manohar, and Kwabena Boahen. Braindrop: A mixed-signal neuromorphic architecture with a dynamical systems-based programming model. *Proceedings of the IEEE*, 107(1):144–164, 2018.

[90] Wolfgang Ocker and Sandra Teger. Hytran: A software system to aid the analog programmer. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, pages 291–298, 1964.

[91] Yakup Paker and Stephen H. Unger. {ADAC} — a programmed direct analog computer. *Mathematics and Computers in Simulation*, 9(1):16 – 23, 1967.

[92] Alberto Paoluzzi, Valerio Pascucci, Michele Vicentino, Claudio Baldazzi, and Simone Portuesi. *Geometric Programming*, pages 51–93. John Wiley & Sons, Ltd, 2005.

[93] Henry M Paynter and Julian Suez. Automatic digital setup and scaling of analog computers. In *Joint Automatic Control Conference*, number 1, pages 156–173, 1963.

[94] Linda Petzold. Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations. *SIAM journal on scientific and statistical computing*, 4(1):136–148, 1983.

[95] Denise R. Ferrier PhD. *Biochemistry (Lippincott Illustrated Reviews Series)*. LWW, 2013.

[96] Eckhard Platen. An introduction to numerical methods for stochastic differential equations. *Acta numerica*, 8(5):197–246, 1999.

[97] Raghu Prabhakar, Yaqi Zhang, and Kunle Olukotun. Coarse-grained reconfigurable architectures. *NANO-CHIPS 2030*, pages 227–246, 2020.

[98] Juan I Ramos. Linearized methods for ordinary differential equations. *Applied mathematics and computation*, 104(2-3):109–129, 1999.

[99] Juan I Ramos. Linearization techniques for singularly-perturbed initial-value problems of ordinary differential equations. *Applied mathematics and computation*, 163(3):1143–1163, 2005.

[100] Harriett Badaker Rigas and David J Coombs. Patch: Analog computer patching from a digital simulation language. *IEEE Transactions on Computers*, 100(10):1140–1146, 1971.

[101] Robert Rihm. Interval methods for initial value problems in odes. *Topics in Validated Computations*, pages 173–207, 1994.

[102] Sylvain Saighi, Yannick Bornat, Jean Tomas, Gwendal Le Masson, and Sylvie Renaud. A library of analog operators based on the Hodgkin-Huxley formalism for the design of tunable, real-time, silicon neurons. *Biomedical Circuits and Systems, IEEE Transactions on*, 5(1):3–19, 2011.

[103] Mitsuji Sampei and Katsuhisa Furuta. On time scaling for nonlinear systems: Application to linearization. *IEEE Transactions on Automatic Control*, 31(5):459–462, 1986.

[104] Sams. Arrangement and scaling of equations. *Mathematics and Computers in Simulation*, 6(3):179 – 182, 1964.

[105] Rahul Sarpeshkar. *Ultra Low Power Bioelectronics: Fundamentals, Biomedical Applications, and Bio-Inspired Systems*. Cambridge University Press, 2010.

[106] Frank Schadt, Friedemann Mohr, and Markus Holzer. Application of kalman filters as a tool for phase and frequency demodulation of iq signals. In *2008 IEEE Region 8 International Conference on Computational Technologies in Electrical and Electronics Engineering*, pages 421–424. IEEE, 2008.

[107] M Schauer and R Heinrich. Quasi-steady-state approximation in the mathematical modeling of biochemical reaction networks. *Mathematical biosciences*, 65(2):155–170, 1983.

[108] Craig R Schlottmann, Samuel Shapero, Stephen Nease, and Paul Hasler. A digitally enhanced dynamically reconfigurable analog platform for low-power signal processing. *IEEE Journal of Solid-State Circuits*, 47(9):2174–2184, 2012.

[109] Christian Schneider and Howard Card. Analog CMOS synaptic learning circuits adapted from invertebrate biology. *Circuits and Systems, IEEE Transactions on*, 38(12):1430–1438, 1991.

[110] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[111] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[112] Kasturi S Shah, Samuel S Pegler, and Brent M Minchew. Two-layer fluid flows on inclined surfaces. *Journal of Fluid Mechanics*, 917, 2021.

[113] Lawrence F Shampine and Charles William Gear. A user's view of solving stiff ordinary differential equations. *SIAM review*, 21(1):1–17, 1979.

[114] LF Shampine. Solving ordinary differential equations for simulation. *Mathematics and Computers in Simulation*, 20(3):204–207, 1978.

[115] Akshay Sharma. *Place and Route Techniques for FPGA Architecture Advancement*. University of Washington, 2005.

[116] Yichen Shen, Nicholas C Harris, Scott Skirlo, Mihika Prabhu, Tom Baehr-Jones, Michael Hochberg, Xin Sun, Shijie Zhao, Hugo Larochelle, Dirk Englund, et al. Deep learning with coherent nanophotonic circuits. *Nature Photonics*, 11(7):441, 2017.

[117] Richard I Sherwood, Tatsunori Hashimoto, Charles W O'Donnell, Sophia Lewis, Amira a Barkal, John Peter van Hoff, Vivek Karun, Tommi Jaakkola, and David K Gifford. Discovery of directional and nondirectional pioneer transcription factors by modeling DNase profile magnitude and shape. *Nature Biotechnology*, 32(2):171–8, mar 2014.

[118] AS Shinde and KC Takale. Study of black-scholes model and its applications. *Procedia Engineering*, 38:270–279, 2012.

[119] Robert D Skeel. Scaling for numerical stability in gaussian elimination. *Journal of the ACM (JACM)*, 26(3):494–526, 1979.

[120] Paul D Smith, Matt Kucic, and Paul Hasler. Accurate programming of analog floating-gate arrays. In *2002 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 5, pages V–V. IEEE, 2002.

[121] Renée St Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. *ACM SIGARCH Computer Architecture News*, 42(3):505–516, 2014.

[122] Marvin L Stein. Automatic digital programming of analog computers. *IEEE Transactions on Electronic Computers*, (2):100–111, 1963.

[123] Marvin L Stein, Jack Rose, and Donn B Parker. A compiler with an analog-oriented input language. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 92–102, 1959.

[124] Luca Sterpone and Massimo Violante. A new reliability-oriented place and route algorithm for sram-based fpgas. *IEEE Transactions on Computers*, 55(6):732–744, 2006.

[125] Vaishali Tehre and Ravindra Kshirsagar. Survey on coarse grained reconfigurable architectures. *International Journal of Computer Applications*, 48(16):1–7, 2012.

[126] Emil J Tejkowski. *ANSIR: a language for patching and checking analog and hybrid Computers*. PhD thesis, University of Notre Dame, 1969.

[127] AEG Telefunken. *Hybrid Precision Analog Computer System RA 770D Operating Manual*. 1966.

[128] Jonathan J. Y. Teo, Sung Sik Woo, and Rahul Sarpeshkar. Synthetic biology: A unifying view and review using analog circuits. *IEEE Trans. Biomed. Circuits and Systems*, 9(4):453–474, 2015.

[129] Naftali Tishby. A dynamical systems approach to speech processing. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 365–368. IEEE, 1990.

[130] Rajko Tomovic. Proceedings of the international association for analog computation method of iteration and analog computation. *Mathematics and Computers in Simulation*, 1(2):60 – 63, 1958.

[131] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[132] Yannis Tsividis. Not your father's analog computer. *IEEE Spectrum*, 55(2):38–43, 2018.

[133] T Turanyi, AS Tomlin, and MJ Pilling. On the error of the quasi-steady-state approximation. *The Journal of Physical Chemistry*, 97(1):163–172, 1993.

[134] Bernd Ulmann. *Analog and Hybrid Computer Programming*. Walter de Gruyter GmbH & Co KG, 2020.

[135] Gustavo Valverde and Vladimir Terzija. Unscented kalman filter for power system dynamic state estimation. *IET generation, transmission & distribution*, 5(1):29–37, 2010.

[136] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.

[137] Sergey Vichik, Murat Arcak, and Francesco Borrelli. Stability of an analog optimization circuit for quadratic programming. *Systems & Control Letters*, 88:68–74, 2016.

[138] Herbert Weiner. The illusion of simplicity: the medical model revisited. *The American journal of psychiatry*, 1978.

[139] Tim Wescott. *Applied control theory for embedded systems*. Elsevier, 2011.

[140] Sung Sik Woo, Jaewook Kim, and Rahul Sarpeshkar. A cytomorphic chip for quantitative modeling of fundamental bio-molecular circuits. *IEEE Trans. Biomed. Circuits and Systems*, 9(4):527–542, 2015.

[141] Sung Sik Woo, Jaewook Kim, and Rahul Sarpeshkar. A digitally programmable cytomorphic chip for simulation of arbitrary biochemical reaction networks. *IEEE transactions on biomedical circuits and systems*, 12(2):360–378, 2018.

[142] Xunzhao Yin, Zoltán Toroczkai, and Xiaobo Sharon Hu. An analog sat solver based on a deterministic dynamical system: (invited paper). In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 794–799, 2017.

[143] Zhou Zongfang and Yong Shi. A convergence of ode method in constrained optimization. *Journal of mathematical analysis and applications*, 218(1):297–307, 1998.

Appendix A

Appendix

$$
\begin{array}{rcl}
\texttt{ival-prop}(x, IA) & = & [x, x] \\
\texttt{ival-prop}(v, IA) & = & \mathit{singleton}(\{x \mid (v, \mathit{ival}) \in A\}) \text{ if } \exists (v, \mathit{ival}) \in IA \\
 & & \textbf{error otherwise} \\
 & & \\
\texttt{ival-prop}(\mathit{binop}(e, e'), IA) & = & \mathit{binop}(\texttt{ival-prop}(e, IA), \texttt{ival-prop}(e', IA)) \\
\texttt{ival-prop}(\mathit{unop}(e), IA) & = & \texttt{ival-prop}(\mathit{unop}(e), IA) \\
\texttt{ival-prop}(\texttt{integ}(e, e')) & = & \textbf{error}
\end{array}
$$

Figure A-1: $\texttt{ival-prop}(e) = \mathit{ival}$ function

A.1 Interval Propagation Function (ival-prop)

Figure A-1 presents the interval function ival-prop, which accepts an expression e and a set of interval assignments $IA \subseteq \mathit{IntervalAssignments} : \mathbb{P}(\mathit{GenericVars} \times \mathit{Intervals})$. The interval function and returns the interval *ival* which captures all the values computed by the expression:

- *ival-prop*(x, IA): The interval of a constant value is $[x, x]$.
- *ival-prop*(v, IA): The interval of a variable v is the interval defined in the dynamical system specification. Each variable interval is identified with a `interval` statement in the DSS.

For the remaining rules, I introduce sub-expressions e and e' with intervals $[x, y] = \texttt{ival-prop}(e)$ and $[x', y'] = \texttt{ival-prop}(e')$.

- *ival-prop*$(\mathit{binop}(e, e'), IA)$: I use interval arithmetic to define the interval of an expression $\mathit{binop}(e, e')$, where *binop* is an mathematical operator which accepts two arguments. This rule covers addition, subtraction, exponentiation, multiplication, minimization ($\texttt{min}(e, e')$), and maximization ($\texttt{max}(e, e')$) functions.

  I implement an interval propagation rule for each mathematical operator. For example, $\texttt{ival-prop}(e - e', IA)$ is $[x - y', y - x']$, where $[x, y] = \texttt{ival-prop}(e, IA)$ and $[x', y'] = \texttt{ival-prop}(e', IA)$, since the smallest possible value is the lower bound of e minus the upper bound of e' and the largest possible value is the upper bound of e minus the lower bound of e'. $\texttt{ival-prop}(e + e', IA)$ is $[x + x', y + y']$ where $[x, y] = \texttt{ival-prop}(e, IA)$ and $[x', y'] = \texttt{ival-prop}(e', IA)$.

- *ival-prop*$(\mathit{unop}(e), IA)$: I use interval arithmetic to derive the interval of an expression $\mathit{unop}(e, e')$, where *unop* is a mathematical operator which accepts one argument.

This rule covers natural logarithms, natural exponentiation (`exp`), absolute value, sign (`sgn`), sine, and cosine functions. This rule also covers the observation (`emit`) and external input (`extvar`) functions.

I implement an interval propagation role for each mathematical operator. For example `ival-prop`$(\texttt{sgn}(e), IA)$ is $[1, 1]$ where $[x, y] =$ `ival-prop`(e, IA) and $x \geq 0$ and $y \geq 0$. If $x < 0$ and $y > 0$, then the returned interval is $[-1, 1]$. If $x < 0$ and $y < 0$, then the returned interval is $[-1, -1]$.

- *`ival-prop`*$(v = \textit{\texttt{integ}}(e, e')$: The interval propagation algorithm does not propagate intervals through integration operations.

$$
\begin{array}{rcl}
\texttt{eval}(x, A) & = & x \\
\texttt{eval}(v, A) & = & \mathit{singleton}(\{x \mid (v, y) \in A\}) \text{ if } \exists (v, y) \in A \\
 & & v \text{ otherwise} \\
 & & \\
\texttt{eval}(e \odot e', A) & = & \texttt{eval}(e, A) \odot \texttt{eval}(e', A) \\
\texttt{eval}(\mathit{func}(e)) & = & \texttt{eval}(e, A) \\
\texttt{eval}(\texttt{integ}(e, e', A) & = & \texttt{integ}(\texttt{eval}(e, A), \texttt{eval}(e', A))
\end{array}
$$

Figure A-2: $\texttt{eval}(e_p) = e_v$ function

A.2 Expression Evaluation Function (eval)

Figure A-2 presents the evaluation function eval, which accepts an expression e and a set of expression-value assignments ($A \subset \mathit{Assignments}$). It evaluates the provided expression by applying the provided assignments.

- $\mathit{eval}(y, A)$: The constant y evaluates to y.
- $\mathit{eval}(v, A)$: The variable v evaluates to its mapped value $\mathit{singleton}((\{x \mid (v, y) \in A\})$ if there is an assignment $(v, y) \in A$. If there is no mapping for variable v, then the evaluation function returns v.
- $\mathit{eval}(e \odot e', A)$: The evaluation function recursively evaluates e and e' to values y and y'. It evaluates $y \odot y'$ and returns the evaluated value. If either or both of these expressions evaluate to expressions (e_{eval} and e'_{eval}), it returns the partially evaluated expression $e_{eval} \odot e'_{eval}$. This rule covers addition, subtraction, exponentiation, multiplication, minimization ($\texttt{min}(e, e')$), and maximization ($\texttt{max}(e, e')$) functions.
- $\mathit{eval}(\mathit{func}(e), A)$: The evaluation function recursively evaluates e to value y. It evaluates $\mathit{func}(y)$ and returns the evaluated value. If the expression e evaluates to an expression (e_{eval}), the algorithm returns the partially evaluated expression $\mathit{func}(e_{eval})$. This rule covers natural logarithms, natural exponentiation (exp), absolute value, sign (sgn), sine, and cosine functions. This rule also covers the observation (emit) and external input (extvar) functions.
- $\mathit{eval}(\mathit{integ}(e, e'), F)$: The evaluation function recursively evaluates the differential equation expression e and initial condition e' to yield the evaluated expressions e_{eval} and e'_{eval}. It returns the evaluated integral $\texttt{integ}(e_{eval}, e'_{eval})$

A.3 Geometric Program Encoding Tricks

The compiler uses several constraint encoding tricks to flexibly implement a wide range of constraints as geometric programming constraints.

Equality over Monomials: Equality relations over monomials can be rewritten into geometric programming constraints of the form $mo = 1$:

$$mo = mo' \Rightarrow \frac{mo}{mo'} = 1$$

The above rewrite rule leverages the fact that the ratio of two monomials is also a monomial.

Inequality over Monomials: Inequalities over monomials can be rewritten into geometric programming constraints of the form $mo \leq 1$:

$$mo \leq mo' \Rightarrow \frac{mo}{mo'} \leq 1$$

The above rewrite rule leverages the fact the ratio of two monomials is also a monomial.

A.3.1 Interval Encoding

The compiler represents interval geometric programming variables as pairs of positive real-valued scalar variables that encode the interval's lower and upper bounds. Because these variables must be positive, the compiler statically reasons over any sign differences during problem generation. The compiler encodes constraints over intervals as constraints over the lower and upper bound variables. The compiler supports $\subseteq$ constraints over numerical intervals to geometric programming constraints. These constraints take the form:

$$mo \cdot ival \subseteq mo'ival'$$

In the above constraint, a scaled numerical interval $mo \cdot ival$ must be contained by the scaled interval $mo'ival'$. The interval $ival$ has bounds $[y_{low}, y_{high}]$ and the interval $ival'$ has bounds $[y'_{low}, y'_{high}]$. T

The interval encoding algorithm analyzes this constraint and transforms it into two constraints over real numbers. The encoding procedure takes advantage of the fact that monomials can only take on positive non-zero values to statically evaluate the $\subseteq$ constraint for cases where the lower and upper bounds of the subinterval and interval have different signs.

TableA.1 presents all the cases handled by the constraint generation. It generates a constraint that restricts the lower bound of the intervals and a constraint that restricts the upper bound of the intervals. I summarize the encoding procedure below:

$y_{low} \geq 0 \wedge y'_{low} \leq 0$: In this case, the lower bound of the subinterval is always greater than the lower

subinterval	interval	constraint
$y_{low} \geq 0$	$y_{low} \geq 0$	$mo_{low} \cdot y_{low} \geq mo'_{low} y'_{low}$
$y_{low} \leq 0$	$y_{low} \leq 0$	$mo_{low} \cdot -y_{low} \leq mo'_{low} - y'_{low}$
$y_{low} \leq 0$	$y'_{low} \geq 0$	false
$y_{low} \geq 0$	$y'_{low} \leq 0$	true
$y_{high} \geq 0$	$y_{high} \geq 0$	$mo \cdot y_{high} \leq mo'_{high} y'_{high}$
$y_{high} \leq 0$	$y_{high} \leq 0$	$mo_{high} \cdot -y_{high} \geq mo'_{high} - y'_{high}$
$y_{high} \geq 0$	$y'_{high} \leq 0$	false
$y_{high} \leq 0$	$y'_{high} \geq 0$	true

Table A.1: Cases for translation of $\subseteq$ operator to geometric programming constraints

bound of the interval. The lower bound constraints, therefore, always hold, and this constraint is trivially true.

$y_{low} \leq \wedge y'_{low} \geq 0$: In this case, the lower bound of the subinterval is always smaller than the lower bound of the interval. The lower bound constraint, therefore, never holds (this constraint is trivially false).

$y_{high} \leq 0 \wedge y'_{high} \geq 0$: In this case, the upper bound of the subinterval is always smaller than the upper bound of the interval. The upper bound constraint, therefore, always holds (this constraint is trivially false).

$y_{high} \geq 0 \wedge y'_{high} \leq 0$: In this case, the upper bound of the subinterval is always larger than the upper bound of the interval. The upper bound constraint, therefore, never holds (this constraint is trivially false).