Accuracy-Aware Optimization of Approximate Programs

by

Saša Misailović

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

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author .... Signature redacted .....
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by ..... Signature redacted .....
Martin C. Rinard
Professor
Thesis Supervisor

Accepted by ..... Signature redacted .....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Accuracy-Aware Optimization of Approximate Programs

by

Saša Misailović

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements of the degree of
Doctor of Philosophy

Abstract

Many modern applications (such as multimedia processing, machine learning, and big-data analytics) exhibit a natural tradeoff between the accuracy of the results they produce and the application's execution time or energy consumption. These applications allow us to investigate new, more aggressive optimization approaches.

This dissertation presents a foundation of program optimization systems that expose and profitably exploit tradeoffs between the accuracy of the results that the program produces and the time and energy required to produce those results. These systems apply *accuracy-aware program transformations* that intentionally change the semantics of optimized programs.

A key challenge to applying accuracy-aware transformations is understanding the *uncertainty* that the transformations introduce into the program's execution. To address this challenge, this dissertation presents program analysis techniques that *quantify* the uncertainty introduced by program transformations. First, this dissertation identifies the properties of subcomputations that are amenable to loop perforation (an accuracy-aware transformation that skips loop iterations). Second, it presents how static analysis can derive expressions that characterize the frequency and magnitude of errors. Third, it presents a system that automatically applies accuracy-aware transformations by formulating accuracy-aware program optimization as standard mathematical optimization problems. The experimental results show that accuracy-aware transformations can help uncover significant performance and energy improvements with acceptable accuracy losses.

Thesis Supervisor: Martin C. Rinard

Title: Professor, Electrical Engineering and Computer Science

Acknowledgments

First of all, I would like to thank my advisor, Martin C. Rinard. Martin provided me with an enormous support throughout my PhD studies, and I had a great privilege to work with him for the past seven years. Martin was enthusiastic about this research direction from the very beginning, and motivated me to persist on this path. Martin would always make himself available to discuss research, and share his technical insights and life advice. Out of the many directives Martin would say to his students to challenge them and cheer them up, the one that I will always remember is "be brilliant!"

I would like to thank my PhD committee members, Professors Saman Amarasinghe, Jonathan Kelner, and Armando Solar-Lezama. In addition to serving on the committee, they have provided me with valuable advice on the content of this thesis and encouraged me throughout my PhD studies. I would also like to thank Professors Arvind, Regina Barzilay, Daniel Jackson, and Youssef Marzouk for their constructive feedback on my research.

This dissertation is a result of a truly collaborative work. I had the great fortune to work with a great team of collaborators on the topic of accuracy-aware transformations, including Hank Hoffmann, Stelios Sidiroglou, Michael Carbin, Deokhwan Kim, Michael Kling, Dan Roy, Sara Achour, Zichao Qi, and Anant Agarwal. Through my interactions with them, I have learned many valuable lessons and skills. I am especially grateful to my long-term collaborators, Hank and Stelios, for their insights and discussions on how to understand loop perforation, and Mike Carbin, for his insights and discussions on how to rigorously analyze accuracy-aware transformations.

I would like to thank Professors Dragan Milicev and Darko Marinov, my undergraduate mentors for believing in me and playing critical role in my decision to pursue my research career. I am especially grateful to Darko for letting me do research in software testing during my final two years of undergraduate school and showing me how rewarding and fun a career in research can be. During this time, I also had the good fortune to collaborate with Professors Viktor Kuncak and Sarfraz Khurshid. All these interactions helped me boost my desire to learn and shaped my research taste.

MIT is proud to be not just a place where people do research, but a vibrant community of individuals driven by curiosity. I had the great fortune of experiencing it firsthand as I met many good friends who have helped me balance my research and life. This includes the "Lambda" group from the 7th floor of Stata, Aleksandar Milicevic, Sachithra Hemachandra, Eunsuk Kang, Harshad Kasture, Jacqueline Lee, Joseph Near, Rishabh Singh, and Jean Yang, who all joined MIT at around the same time as I did, and with whom I enjoyed spending time and have remained friends with ever since. I am grateful to my friends from MIT and Boston who have made me feel welcome here, including Sara Achour, Milos Cvetkovic, James Cowling, Michal Depa, Zoran Dzunic, Neha Gupta, Fredrik Kjolstad, Ivan Kuraj, William Li, Danilo Mandic, Velibor Misic, Sudeep Pillai, Nadia Polikarpova, Derek Rayside, Kuat Yessenov, Jungbum Yung, Gabriel Zaccak, and many others. I would like to thank my friends Milos Gligoric, Tihomir Gvero, and Tatjana Petrov for occasionally stopping by MIT to discuss research and life in grad school. I also want to thank my officemates and other members of Martin's group, including Vijay Ganesh, Michael Gordon, Fereshte Khani, Fan Long, Jeff Perkins, Jiasi Shen, Karen Zee, and others for the enjoyable working environment and great research and life-related conversations.

I am grateful to many people from MIT community for helping with logistics during my studies. I do not have enough words to thank our administrator Mary McDavitt, who has always been there to help me with all sorts of budgetary, organizational, and travel questions I had over the years. I would like to thank Lary Hardesty for continuously covering my research in MIT News and Marilyn Levine for helping me become a better writer. I am grateful to Janet Fischer who was always patient with all of my last-minute submissions of paperwork and theses.

Parts of the research presented in this dissertation were funded by grants and projects from NSF, DOE, and DARPA.

Finally, none of this would have been possible without the unconditional love and support from my parents, Dragomir and Mileva Misailovic. They have always been by my side, believing in me and helping me overcome challenges and achieve my goals. I will always remain grateful to them. And to them I dedicate this dissertation.

Contents

List of Figures

List of Tables

1 Introduction

Modern applications are expected to run fast. Many emerging applications, such as multimedia processing, machine learning, and big data analytics, operate on noisy inputs, large data sets, or solve computationally intensive problems with multiple acceptable solutions. The developers of these applications have the freedom to intentionally trade some of the accuracy of the application's result in return for faster execution. For instance, the main task of a video encoder is to compress streams of raw video frames. To achieve the desired level of compression, video processing researchers and practitioners have devised various approximate compression algorithms that produce videos with acceptable quality [14].

Modern hardware architectures are expected to be energy-efficient. For a long time, standard circuit-design techniques have successfully scaled the voltage and the size of the circuits, while maintaining reliable operation. However, these techniques are reaching their limit (also known as the end of Dennard scaling [41]). To enable their hardware architectures to operate efficiently, hardware designers have proposed various designs of on-chip components, memories, and accelerators that trade reliability and accuracy of their operations for reduced circuit size and energy consumption [42, 43, 66, 67, 85, 89].

Modern computer systems are expected to be resilient to faults. The standard fault-tolerance mechanisms implement expensive reexecution or replication techniques to achieve resiliency. However, modern system infrastructures, such as Google's MapReduce, have offered the ability to continue the execution of a computation even if some of the software or hardware components become unresponsive or experience fatal errors [34]. As a result, the application continues its execution to produce a partial and/or approximate result, instead of no result at all.

Despite the prominent role of approximation in applications, architectures, and systems, standard program analysis and compilation systems do not take the advantage of these approximation opportunities. The traditional approaches to program optimization aim to preserve program semantics and are, therefore, too rigid to exploit the full optimization potential of the applications. This leaves a software developer solely responsible for managing all aspects of approximation, which often results in inflexible computations with approximation choices hard-coded in the implementation of the computation.

1.1 Accuracy-Aware Program Transformations

To enable flexible choices for approximating applications and automate the optimization of these applications, researchers, including the author of this dissertation, have proposed compiler-level *accuracy-aware transformations*. These transformations intentionally change the semantics of programs to trade accuracy for improved performance, energy consumption, or resilience by exploiting the properties of program's inputs, structure, and execution environment.

Performance-oriented transformations. These transformations reduce the amount of work that a program performs [25, 53, 71, 75, 79, 80, 95, 96, 102, 113, 125]. For instance, loop perforation is an accuracy-aware transformation that causes a program to skip iterations of `for` loops [79, 113]. Loop perforation can cause the loop such as `for (int i = 0; i < n; i++) {...}` to execute only a subset of its iterations. For instance, the transformation can cause the loop to execute only half of iterations by changing the induction variable increment from `i++` to `i+=2`, or by changing the loop bounds check from `i < n` to `i < n/2`.

Energy-oriented transformations. These transformations instruct a program to use hardware components that aggressively save energy by allowing errors in the results of their operations [19, 43, 73, 105]. For instance, these hardware platforms can provide exact and approximate standard arithmetic operations and reliable and unreliable memories. Annotations of the standard operator symbols, such as "`+.`", specify that arithmetic operations execute approximately. Annotations of declarations, such as "`int x in urel`", specify that variables can be stored in approximate memories [19, 73].

Resiliency-oriented transformations. These transformations allow a program to continue executing through otherwise fatal errors to produce at least a part of the original result [18, 58, 97, 112]. For instance, infinite loops cause programs to be unresponsive; to regain program responsiveness, Bolt can detect whether the loop is infinite and if so, transforms the running program to continue the execution past the infinite loop and produce at least a part of its result [18, 58].

1.2 Accuracy-Aware Program Optimization

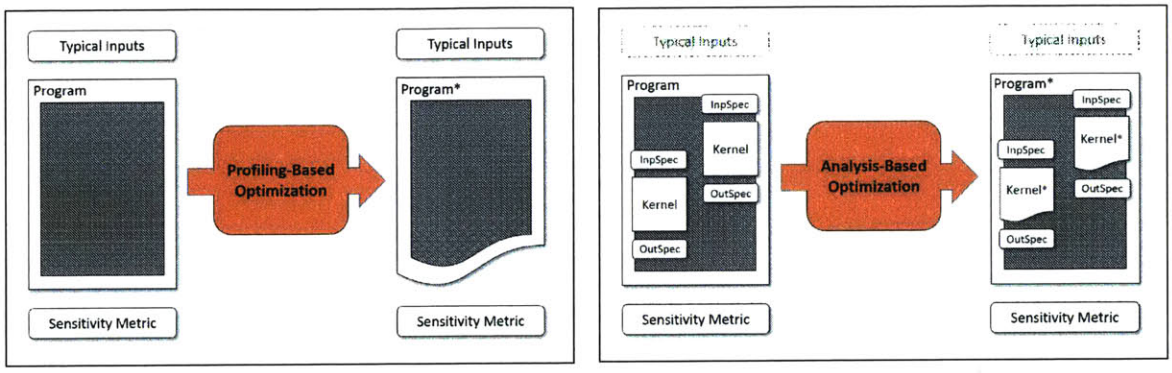
Automatically optimizing approximate programs using accuracy-aware transformations provides new opportunities for reducing engineering effort and resource consumption and in-

creasing program resilience. This opportunity, however, comes at a price – the transformations introduce uncertainty into the program's execution, which reflects on the quality of the results it produces. This uncertainty raises a number of new research questions: 1) how to identify parts of a program that are good candidates for accuracy-aware transformations; 2) how to characterize the effects of a transformation on the program's execution, especially the result's accuracy; and 3) how to automatically discover transformations that provide maximum performance gains for acceptable accuracy losses.

The early techniques for accuracy-aware optimization (such as those we previously proposed [53, 75, 79, 95, 96]) have used a ***sensitivity profiling-based*** approach to answer these questions. These techniques require a developer to provide a set of representative inputs and an application-level sensitivity metric that quantifies the accuracy of the produced result (e.g., peak signal-to-noise for video encoders). Then, a sensitivity profiler applies the accuracy-aware transformations at various points in the program and validates the transformations by testing whether the transformed programs, when executed on the provided inputs, produce results with acceptable accuracy (as calculated by the sensitivity metric). While these techniques are effective in finding transformed programs with attractive tradeoffs, they do not provide accuracy guarantees. Specifically, since this approach relies only on the representative inputs provided by the developer, its results do not generalize and do not provide guarantees for other inputs.

In contrast to these purely dynamic optimization approaches, this dissertation investigates a novel ***analysis-based*** approach that combines static program analysis with mathematical optimization techniques to provide a foundation for rigorous program optimization using accuracy-aware transformations. This approach operates on time-consuming subcomputations (that we call approximate kernels) for which a developer provides a formal specification of the kernel's inputs and the expected output accuracy. Our program analysis can ensure that the approximate version of the kernel satisfies the probabilistic output specification for all inputs that adhere to the input specification. The program optimization algorithm can use this analysis to reduce the kernel approximation to a standard mathematical optimization problem. This approach does not require representative inputs, but a developer can optionally use sensitivity profiling to 1) help identify approximate kernel computations and 2) derive the kernel-level accuracy specifications that likely satisfy the application-level sensitivity metric.

Figure 1.1 illustrates the conceptual difference between the two approaches. The profile-based optimization transforms program subcomputations driven by the inputs and subject to the application-level sensitivity metric. While it can often find attractive tradeoffs, it



(a) Profile-Based Optimization

(b) Analysis-Based Optimization

Figure 1.1: Approaches for Accuracy-Aware Optimization

does not provide an intuition for why the transformations work and how we can quantify the effect of the changes (illustrated by a shaded area of the program). In contrast, the analysis-based optimization operates on explicitly exposed approximate kernels with their specifications, which the developer wrote to meet the application-level accuracy requirement (optionally derived using sensitivity profiling). Then, the rigorous analysis and optimization techniques can automatically approximate the kernel functions, while satisfying the developer's specifications. This approach can, therefore, improve a developer's understanding of why accuracy-aware transformations work. In the rest of this section, we present the main components of the analysis-based accuracy-aware optimization.

1.2.1 Approximate Kernels

Many approximate computations have a specific structure. A majority of their work is performed in one or several *approximate kernel computations*. Each execution of a kernel computation typically processes a part of the application's input and either directly produces a part of the application's output or guides the execution of the application to produce the final output. Transforming the kernels and modifying their results has the potential to only slightly reduce the accuracy of the application's final output.

An instance of an approximate kernel is a loop that iterates over a list of elements and aggregates the elements to produce a final sum:

```
float sum = 0;
for (int i = 0; i < n; i++)
    sum = sum + a[i];
```

This loop computes the sum `sum` by aggregating the elements of an array `a` with `n` elements. In this dissertation, we identify that a number of approximate kernels have similar structure and functionality and show that they are instances of computational patterns amenable to approximation. We divide patterns by their structural properties (e.g., a kernel loop calculates a sum) or functional properties (e.g., a kernel loop's result is used as a distance metric within the application).

We can apply multiple transformations to the summation loop. For instance, to run the loop faster, loop perforation can change the induction variable increment from `i++` to `i+=2`. To save energy when executing the loop body on approximate hardware, a compiler can 1) transform the addition operator in the expression `sum = sum + a[i]` to its approximate version `sum = sum +. a[i]` or 2) specify that the array `a` should be stored in unreliable memory using a declaration `float[] a in urel` [19, 73]. Each of these transformations can cause the loop to produce a result `sum` that differs from the result of the original loop. To quantify this difference, we use static program analysis.

1.2.2 Analyzing Approximate Kernel Transformations

The motivation for our analysis approach rests on the empirical observation that the results of many transformed approximate kernels exhibit *small deviations* from the results of the original kernels *most of the time* [25, 27, 77, 78, 113]. Therefore, this dissertation investigates the set of program analyses of approximate kernels that characterize how accuracy-aware transformations affect the accuracy of the kernel's result.

We split the accuracy analysis of accuracy-aware transformations into two subproblems: 1) analyze how the transformation affects the results of the kernel computations and 2) analyze how the error propagates through the execution. For the first subproblem, this dissertation presents a program analysis-based approach for optimizing the computations that satisfy the developer's accuracy specification. For the second subproblem, this dissertation shows how sensitivity analysis can help developer identify the kernels and derive accuracy specification. To run the analysis, a developer provides the specification of typical inputs and the specification of output accuracy for the kernel.

Kernel Input Specification. The input specification contains intervals or distributions of the kernel's inputs. This specification characterizes the developer's level of knowledge about the inputs. For instance, a developer can specify that the elements of the array `a` in the example computation belong to the interval $[0, 10]$ or that the elements have Gaussian $N(0, 1)$ distribution. Specifications can also be relational – for instance, the elements of the

array a in the approximate program execution have the same value as the elements in the exact program execution with probability at least 0.99.

Output Accuracy Specification. The output specification is a probabilistic assertion about the output of the computation, which checks for a typical error of the computation. The assertions are relational in that they compare the outputs of the original and approximate executions. They have a general form: "*Assuming that the inputs have the specified properties, the absolute difference between the results of the original and transformed program is less than or equal to* Δ *with probability at least* p." The developer provides the numerical quantities Δ and p. The special cases of these assertions consider only the magnitude of numerical error when $p = 1$ and the frequency of producing an incorrect result, when $\Delta = 0$.

Accuracy Analysis. Analyses operate on approximate kernels that are amenable to accuracy-aware transformations. Given the input specification for a kernel, the analyses compute expressions that represent the error induced by the transformation. The analyses then check whether these expressions satisfy the output accuracy specifications.

This dissertation presents analysis of representative accuracy-aware transformations. The analysis of loop perforation operates on a set of structural computational patterns amenable to perforation. The analysis expects that the input specification provides a distribution of the inputs of the computation. For instance, to analyze the effect of loop perforation, a developer can specify that all elements of the array a come from a Gaussian $N(0, 1)$ distribution. The accuracy assertion specifies that the difference between the sum variable computed by the original version of the summation computation is smaller than 0.5 with probability at least 0.99. The analysis calculates the accuracy expression (parameterized by the amount of skipped loop iterations) for the summation structural pattern and checks whether it satisfies the developer's accuracy assertion.

The analysis of approximate arithmetic operations and data stored in approximate memories operates on functions with scalar and array assignments, conditionals and finite loops. For instance, a developer's specification may state that for any input the kernel computation can provide the correct result with probability at least 0.99. The analysis computes the expressions for the probability that the operations within the kernel executed correctly (parameterized by the choice of approximate operations and data stored in approximate memory) and checks whether it satisfies the developer's output specification.

1.2.3 Searching for Approximate Kernel Transformations

In contrast to previous approaches for optimizing programs with accuracy-aware transformations that generate transformed versions of kernels and check whether they satisfy the developer's accuracy specification [53, 75, 79, 95, 96], this dissertation presents the first system that uses the results of a probabilistic accuracy analysis to reduce the problem of selecting the accuracy-aware transformations in approximate kernels to a standard mathematical optimization problem.

This system, called Chisel, formulates the placement of approximate arithmetic operations and placement of data in approximate memories as an integer linear optimization problem. Chisel's sound static analysis generates inequalities that constrain the probability that the function executes with acceptable accuracy. Chisel's optional sensitivity profiler helps a developer derive the accuracy specification for the kernel. Chisel also generates an objective function that estimates energy savings from execution traces of the function on representative inputs. All constraints and the objective function are functions of the configuration vector that encodes the choices for approximating instructions and variables. Chisel then dispatches the constructed optimization problem to an off-the-shelf integer optimization solver to find the optimal configuration vector. Finally, Chisel uses the computed configuration vector to place approximate instructions and data in the kernel. This approximate version of the kernel is guaranteed to satisfy its output specification.

1.3 Problem Statement

The described techniques for analysis and optimization of approximate kernel computations comprise the foundation of a rigorous framework for accuracy-aware optimization. The goal of such accuracy-aware program optimization framework is to automatically find *profitable* and *controllable* accuracy/performance tradeoffs. A tradeoff is profitable if the approximate computation executes faster than the original computation, with acceptable accuracy losses. A tradeoff is controllable if the accuracy of the approximate result satisfies the developer's accuracy specification.

By constructing the main components of the accuracy-aware optimization framework, this dissertation therefore investigates the following main hypothesis:

> *Systems that automatically transform programs to trade accuracy for performance and/or energy guided by program analysis can obtain* ***profitable*** *and* ***controllable*** *tradeoffs*

The dissertation presents the result of our investigation of the main hypothesis:

Chapter 2. This chapter presents a quantitative and qualitative analysis of approximate kernels in real-world benchmark applications. To identify and analyze the approximation of these kernel computations, we use a sensitivity profiler that applies loop perforation and extends our previous SpeedPress approximate compiler [79]. Our quantitative evaluation shows that transforming one or several approximate kernels can substantially improve the performance of the benchmark applications. The evaluation found the maximum observed performance improvements of over six times for a 10% output quality loss. Our qualitative analysis of approximated computations identifies several functional and structural patterns of approximated computational kernels. We also identified that the transformed computations rarely experience large errors. The results presented in this chapter guide our design of rigorous analysis and optimization of approximate computations.

Chapter 3. This chapter formally represents loop perforation and presents a probabilistic analysis of computations transformed using loop perforation. We present an analysis for four structural code patterns including summation, averaging, ratio, or finding a value with minimum/maximum score computed within the loop. The randomness that the analysis quantifies comes from the inputs – a developer specifies the probability distribution of the inputs and/or intermediate values that the computation produces. Given this specification, the analysis computes expressions for the expected error, variance, and the probability of large errors as functions of the fraction of the skipped loop iterations.

Chapter 4. This chapter introduces Chisel, a system for accuracy-aware and reliability-aware optimization of kernel functions that run on approximate hardware. We also present the formalization of an approximate hardware system that consists of a processor with an approximate ALU and cache memory, and an approximate main memory.

Chapter 5 This chapter presents Chisel's optimization algorithm. We present the foundations of the accuracy analysis that quantifies the frequency and magnitude of the kernel's error induced by the transformations. We show how Chisel can use the analysis results to construct a mathematical optimization problem to automatically generate an approximate kernel function that executes with minimum energy consumption while satisfying the developer's specification.

Chapter 6 This chapter presents the evaluation of Chisel's optimization algorithm. Our evaluation shows that Chisel was able to optimize approximate functions in five programs from the image processing and financial analysis domains to obtain system-level energy

savings of up to 20% on a set of accuracy/energy specifications of several approximate hardware designs while preserving reliability guarantees.

Chapter 7. This chapter presents related work, which includes research on software-level and hardware-level approximate computing and related areas such as probabilistic programming, program analysis of emerging computations, and approximate database queries.

Chapter 8. This chapter presents directions for future research that can extend accuracy-aware optimization and advance the general area of approximate computing.

1.4 Contributions

This dissertation makes the following main contributions:

Rigorous Accuracy-Aware Optimization. It presents an optimization framework for analyzing and optimizing programs transformed using accuracy-aware transformations. Our analysis-based approach operates on approximate kernel computations and provides guarantees that the transformed computations satisfy developer-provided accuracy specifications.

Transformation and Accuracy Specification. It presents a formalization of several representative accuracy-aware transformations. This set of transformations includes loop perforation, approximate arithmetic instructions, and storing data in approximate memories. We also present specifications that enable a developer to specify probabilistic constraints that characterize the accuracy of approximate kernel computations and the properties of their inputs.

Accuracy Analysis of Approximate Kernels. It presents a set of static accuracy analyses for accuracy-aware transformations. These analyses use probabilistic reasoning to quantify the magnitude and frequency of the differences between the results of the original and transformed versions of the approximate kernels.

Accuracy-Aware Optimization Algorithm. It presents Chisel, the first system that reduces the problem of selecting accuracy-aware transformations in programs to a mathematical optimization problem (in particular, integer linear programming). It therefore provides a novel strategy for efficiently exploring the tradeoff space and finding optimal accuracy/performance tradeoffs.

2 Characterization of Approximate Kernels Using Loop Perforation

This chapter investigates properties of transformed approximate computations. We perform a quantitative and qualitative analysis of programs transformed using loop perforation (an accuracy-aware transformation that skips loop iterations).

Our analysis is based on *sensitivity profiling*. This is a dynamic program analysis that generates approximate programs by using accuracy aware transformations and checks whether these computations produce results that satisfy the developer's accuracy specification. This chapter presents sensitivity profiling that extends the SpeedPress compiler framework, which we previously constructed to transform programs using loop perforation [79]. Section 2.1 presents the overview of the sensitivity profiling.

We characterize the approximate execution of programs by transforming seven applications from the PARSEC benchmark suite [11] using loop perforation. Section 2.2 presents the approximate benchmarks, their accuracy specifications, and representative inputs used in the experiments. This chapter presents several results of our investigation:

Performance and Accuracy of Perforated Computations. Our results show that loop perforation can help significantly improve performance for five of the seven applications. In each of these applications, our evaluation identified a single loop or a small number of loops that are good candidates to loop perforation.

Specifically, our performance measurements show that the perforated applications can run as much as seven times faster than the original applications while producing outputs that differ by less than 10% from the corresponding outputs of the original applications. Our empirical results also show that the perforated programs produce outputs with similar level of accuracy even for a different set of inputs (and not just those used to guide sensitivity profiling). Section 2.3 presents the results of quantitative analysis for loop perforation.

Structure and Functionality of Perforated Kernels. We have identified computational patterns that interact well with loop perforation. Specifically, we have identified structural patterns, which indicate that loop perforation works well in cases when the loops have a specific structure (e.g., computing a sum of data items computed in each loop iteration). We have also identified functional patterns, which indicate how the rest of the computation uses the subcomputation's results. Overall, these results indicate that successful perforations target computations that are partially redundant at both *the local level* of a loop and *the global level* of applications. Section 2.4 presents the description of functional and structural computational patterns 1) that we identified in our evaluation of the approximate benchmarks and 2) that are amenable to loop perforation. Section 2.5 further presents a detailed description of the instances of these patterns (i.e., computations that loop perforation successfully transformed) in the benchmark applications.

Frequency and Magnitude of Perforated Kernel's Error. Our results show that the approximate kernels that belong to the identified computational patterns, when transformed, rarely produce large absolute errors (compared to the original kernel). We observed that the likelihood of large errors increases as we apply more aggressive transformations, but even then, the errors are typically much smaller than the maximum error. Section 2.6 presents an empirical study of error magnitude and frequency of perforated computations.

Overall, these results give us useful insights that can help us build a rigorous analysis of approximate computations. We discuss these insights in Section 2.7.

Sources. The previous version of the research presented in this chapter appeared in [113]. Section 2.6 is based on research previously presented in [77]

2.1 Sensitivity Profiling in SpeedPress

SpeedPress [79, 113] is a compiler framework that automates program optimization with loop perforation. SpeedPress uses a profiling-based approach to explore the tradeoff space and discover loops that maximize the performance of the transformed program while satisfying the accuracy specification for a set of representative inputs.

As input to SpeedPress, a developer provides the original application, a set of representative inputs, and a specification of the program's accuracy. We describe the parts of the

specification in Section 2.1.1. SpeedPress operates in three steps. In the first step, SpeedPress performs standard performance profiling to find loops in which the program spends the majority of the time. In the second step, SpeedPress finds the set of time consuming loops that can be perforated by generating approximate programs and checking if they satisfy the developer's accuracy specification. SpeedPress uses loop perforation (Section 2.1.2) as its transformation. In the third step, SpeedPress perforates multiple loops at the same time and constructs a *Pareto-optimal tradeoff curve*, which contains the approximate programs that exhibit the most profitable tradeoffs between performance and accuracy. We describe these steps in Section 2.1.3.

2.1.1 Developer's Specification

The program-level developer's accuracy specification consists of three components:

Output Abstraction. The output abstraction is a function that works with the program's output (and optionally its input) to compute a value or a list of values that represent relevant properties of the output. We denote a result of output abstraction as $\boldsymbol{o} = (o_1, \ldots, o_m)$.

The output abstraction function is application-specific and is provided by the developer. Typically, an output abstraction function selects relevant numbers from an output file or files or computes an application-specific measure of the quality of the output. Many approximate computations come with such quality measures already defined and available (e.g., as a part of program testing). We present examples of output abstractions in Section 2.2.

Sensitivity Metric. The sensitivity metric function computes the distance between the results of the original and transformed programs. The sensitivity function $Q(\boldsymbol{o}, \hat{\boldsymbol{o}})$ takes as input the two lists of values computed by the output abstraction function. The abstracted output $\boldsymbol{o}$ comes from the execution of the original program. The abstracted output $\hat{\boldsymbol{o}}$ comes from the execution of the transformed program.

The function $Q(\cdot, \cdot)$ is also specified by the developer. We will typically use a distance function based on the relative difference, called *distortion* [95]. In particular, distortion is a weighted mean scaled difference between the output abstraction components $\boldsymbol{o} = (o_1, \ldots, o_m)$ from the original program and the output abstraction components $\hat{\boldsymbol{o}} = (\hat{o}_1, \ldots, \hat{o}_m)$ from the perforated program:

$$Q(\boldsymbol{o}, \hat{\boldsymbol{o}}) = \frac{1}{m} \sum_{i=1}^{m} w_i \left| \frac{o_i - \hat{o}_i}{o_i} \right| \tag{2.1}$$

Each weight w_i captures the relative importance of the i-th component of the output abstraction. Note that the closer this sensitivity metric Q is to zero, the more accurate the transformed program is. We often report sensitivity metrics as percentages.

For multiple inputs, the developer can select the overall sensitivity of the transformed program as an average or as a maximum sensitivity out of those computed for each input.

Sensitivity Goal. A developer can specify the bound b that indicates the maximum acceptable (tolerable) accuracy loss. Specifically, the bound represents the extreme value of the sensitivity metric. It can be a maximum or a minimum, which depends on whether the value 0 of the metric $Q(\cdot, \cdot)$ represents the maximum or the minimum accuracy of the result. For instance, for the sensitivity metric Q from Equation 2.1, the program executes acceptably if $Q(\boldsymbol{o}, \hat{\boldsymbol{o}})$ is less than b (since the distortion value of 0 indicates fully accurate output). On the other end, if infinity (or other large value) is the maximum of the sensitivity metric, then the program executes if $Q(\boldsymbol{o}, \hat{\boldsymbol{o}})$ is greater than b.

2.1.2 Loop Perforation Transformation

SpeedPress implements the loop perforation transformation within the LLVM compiler framework [64]. The perforation pass works with any loop that the existing LLVM loop canonicalization pass, *loop-simplify*, can convert into the following form:

```
for (i = 0; i < M; i++) { ... }
```

In this form, the loop has a unique induction variable (in the code above, `i`) initialized to `0` and incremented by `1` on every iteration, with the loop terminating when the induction variable `i` exceeds the bound (in the code above, `M`). The class of loops that LLVM can convert into this form includes, for example, `for` loops that initialize an induction variable to an arbitrary initial value, increment the induction variable by an arbitrary constant value on each iteration, and terminate when the induction variable exceeds an arbitrary bound.

The loop perforation transformation takes as a parameter a loop perforation rate r, which represents the expected percentage of loop iterations to skip. Interleaving perforation transforms the loop to perform every `n`-th iteration (here the perforation rate is r = `1-1/n`). Conceptually, the perforated computation looks like:

```
for (i = 0; i < M; i += n) { ... }
```

In addition to interleaving perforation, SpeedPress can also apply other types of perforation. Truncation perforation skips a contiguous sequence of iterations at either the beginning or

the end of the loop. For example, it can replace the loop condition `i < M` with `i < M/n`. Random perforation randomly skips loop iterations. Perforated computations can also skip only one out of `M` iterations. Our previous work [52] presents a detailed treatment of how to implement various loop perforation strategies.

2.1.3 Sensitivity Profiling Algorithm

The loop perforation space exploration algorithm takes as input an application, an accuracy specification for that application, a set of training inputs, and a set of perforation rates. The algorithm produces a set S of loops to perforate at specified perforation rates.

Sensitivity Profiling for Individual Loops

The exploration algorithm starts with a set of candidate loops. The algorithm can be configured to consider only loops that execute for more than a certain percentage of the execution time. In general, perforating a candidate loop may cause the program to crash, generate unacceptable output, produce an infinite loop, or decrease its performance. Algorithm 2.1 is designed to find and remove such *critical* loops from the set of candidate loops. The algorithm perforates each loop in turn, using each of the specified perforation rates, then runs the perforated program on the training inputs.

The sensitivity profiling algorithm filters out a loop if its perforation (1) fails to improve the performance as measured by the speedup s, which is the execution time of the perforated application divided by the execution time of the original unperforated program running on the same input, (2) causes the application to exceed the sensitivity bound b or, (3) introduces memory errors (such as out of bounds reads or writes, reads to uninitialized memory, memory leaks, double frees, etc.). If a memory error causes the execution to crash on some input t, its sensitivity a_t is ∞. The result of sensitivity profiling is the set of perforatable loops $P = \{\langle l, r\rangle\}$, where $\langle l, r\rangle$ specifies the perforation of loop l at rate r.

Perforation Space Exploration

To find the potential for how much an application can be perforated, the algorithm for exploring the space of perforated programs tests programs in which it perforates all combinations of perforatable loops using sensitivity profiling.

The algorithm starts with the set of perforatable loops, then exhaustively explores all combinations of perforatable loops l at their specified perforation rates r. The algorithm executes all combinations on all training inputs and records the resulting speedup and

Inputs:
A - an application
T - a set of representative inputs
Q - a sensitivity metric
b - a sensitivity bound
L - a set of candidate loops for perforation
R - a set of perforation rates
Outputs: P - a set of loops and perforation rates for A that satisfy the developer's accuracy specification.

$P = \emptyset$
for $t \in T$ **do**
  Run A on t, record execution time e_t and output abstraction $\boldsymbol{o}_t$
end for
for $\langle l, r \rangle \in L \times R$ **do**
  Let $A_{\langle l,r \rangle}$ be A with l perforated at rate r
  for $t \in T$ **do**
    Run $A_{\langle l,r \rangle}$ on t, record execution time $\hat{e}_t$ and output abstraction $\hat{\boldsymbol{o}}_t$
    $a_t = Q(\boldsymbol{o}_t, \hat{\boldsymbol{o}}_t)$ and $s_t = e_t / \hat{e}_t$.
  end for
  $\overline{s} = (\sum_{t \in T} s_t)/||T||$; $\overline{a} = (\sum_{t \in T} a_t)/||T||$
  if $\overline{a} < b \wedge \overline{s} > 1$ **then**
    for $t \in T$ **do**
      Run $A_{\langle l,r \rangle}$ using Valgrind to find E_t (memory errors)
    end for
    if $\bigcup_{t \in T} E_t = \emptyset$ **then**
      $P = P \cup \{\langle l, r \rangle\}$
    end if
  end if
end for
return P

Figure 2.1: Sensitivity Profiling Finds the Set of Perforatable Loops P in Application A Given Training Inputs T, Sensitivity metric Q and Sensitivity Goal b.

accuracy. It also runs each perforated candidate program under Valgrind [86], discarding the combination if Valgrind detects a memory error (similar to sensitivity profiling algorithm for individual loops).

We use the results of exploration to compute the set of Pareto-optimal perforations in the induced performance vs. accuracy tradeoff space. A perforation is *Pareto-optimal* if there is no other perforation that provides both better performance and better accuracy. A user of the analysis can provide a sensitivity bound b to obtain the perforated program whose sensitivity is the closest below b, and provides the maximum speedup.

This approach is feasible for applications (such as those in our set of benchmarks) that spend the majority of their time in relatively few loops. If the application has enough loops to make exhaustive exploration infeasible, it is possible to either use a greedy algorithm (e.g., as one in [79]) or drop enough of the least time-consuming perforatable loops to make exhaustive exploration feasible. Hybrid approaches are also possible.

2.2 Benchmarks and Inputs

Benchmark	Training Inputs	Production Inputs	Source
x264	4 HD videos of 200+ frames	12 HD videos of 200+ frames	PARSEC & videos from xiph.org [70]
bodytrack	sequence of 100 frames	sequence of 261 frames	PARSEC & additional input provided by benchmark authors
swaptions	64 swaptions	512 swaptions	PARSEC & randomly generated swaptions
ferret	256 image queries	3500 image queries	PARSEC
canneal	4 netlists of 2M+ elements	16 netlists of 2M+ elements	PARSEC & additional inputs provided by benchmark authors
blackscholes	64K options	10M options	PARSEC
streamcluster	4 streams of 19K-100K data points	10 streams of 100K data points	PARSEC & UCI Machine Learning Repository [8]

Table 2.1: Summary of Training and Production Inputs

Table 2.1 summarizes the sources of the evaluation inputs. We evaluate loop perforation using a set of benchmark applications from the PARSEC 1.0 benchmark suite [11]. These applications were chosen to be representative of modern and emerging workloads for the next generation of processor architectures. We use the following applications: x264, bodytrack, swaptions, ferret, canneal, blackscholes, and streamcluster. Together these benchmarks represent a broad range of application domains including financial analysis, media processing, computer vision, engineering, data mining, and similarity search.

For each benchmark we acquire a set of evaluation inputs, then pseudorandomly partition the inputs into training and production sets. We use the training inputs to drive the loop perforation space exploration algorithm (Section 2.1.3) and the production inputs to evaluate how well the resulting perforations generalize to unseen inputs. For each benchmark, the PARSEC benchmark suite contains "native" inputs designed to represent the inputs that the application is likely to encounter in practical use. We always include these inputs in the set of evaluation inputs. For many of the benchmarks we also include other representative inputs, typically to increase the coverage range of the evaluation set, to promote an effective partition into reasonably-sized training and production sets, or to compensate for deficiencies in the native inputs provided with the PARSEC benchmark suite.

For each of the benchmarks we describe below what the application does, what inputs we executed, and how we defined the sensitivity metric of the computation.

x264. This media application performs H.264 encoding on raw video data. It outputs a file containing the raw (uncompressed) input video encoded according to the H.264 standard. The output abstraction extracts the peak-signal-to-noise ratio (PSNR) (which measures the quality of the encoded video relative to the original, unencoded video) and the bitrate (which measures the compression achieved by the encoder). The sensitivity metrics weighs both PSNR and bitrate equally with a weight of one. If the reference decoder fails to parse the encoded video during training, we record the sensitivity metric of 100% and reject the perforation.

Since the native PARSEC input contains only a single video, we augment the evaluation input set with additional inputs from xiph.org [70]. These inputs represent the standard test videos that are used by developers of software that manipulates video files.

Bodytrack. This computer vision application uses an annealed particle filter to track the movement of a human body [35]. It produces two output files: a text file containing a series of vectors representing the positions of the body over time and a series of images graphically depicting the information in the vectors overlaid on the video frames from the cameras. The output abstraction extracts vectors that represent the position of the body. The weight of each vector in the sensitivity metric is proportional to its magnitude (in the result of the original program). Vectors which represent larger body parts (such as the torso) therefore have a larger influence on the sensitivity metric than vectors that represent smaller body parts (such as forearms).

The application requires data collected from carefully calibrated cameras. The native PARSEC input contains a single sequence of 261 frames. We augment the evaluation input

set with another sequence of 100 frames that we obtained from the maintainers of the PARSEC benchmark suite.

Swaptions. This financial analysis application uses a Monte Carlo simulation to solve a partial differential equation that prices a portfolio of swaptions. The output abstraction simply extracts the swaption prices. The sensitivity metric computes the distortion between the extracted prices.

Each input to this application contains a set of parameters for each swaption. The native PARSEC input simply repeats the same parameters multiple times, causing the application to repeatedly calculate the same swaption price. We therefore augment the evaluation input set with additional randomly generated parameters so that the application computes prices for a range of swaptions.

Ferret. This application performs a content-based similarity search of an image database. For each input query image, ferret outputs a list of similar images found in its database. The sensitivity metric computes the degradation of precision of Ferret's search. To compute the sensitivity metric, the procedure first calculates the intersection of the sets of images returned by the perforated and original versions, then computes recall by dividing the size of this set by the size of the set of images returned by the original version, then subtracting this number from one. So, if both versions return 10 images, 9 of which are the same, the sensitivity metric is 0.1.

The PARSEC benchmark suite contains 3500 different image queries. We do not use additional inputs.

Canneal. This engineering application uses simulated annealing to minimize the routing cost of a microchip design. Canneal prints a number representing the total routing cost of a netlist representing the chip design; we use this cost as the output abstraction and distortion as the sensitivity metric. The resulting accuracy specification directly captures the application's ability to reduce routing costs.

The PARSEC benchmark contains only one large netlist. We obtained additional input netlists from the maintainers of the PARSEC benchmark suite.

Blackscholes. This financial analysis application solves a partial differential equation to compute the price of a portfolio of European options. The PARSEC application produces no output. We therefore modified the application to print the option prices to a file. The output abstraction extracts these option prices. The distortion calculation weights these prices equally (with a weight of one). The resulting accuracy specification directly captures the ability of the perforated application to compute accurate option prices.

The native input from the PARSEC benchmark suite contains 10 million different option parameters. We do not augment this input set with additional inputs.

Streamcluster. This data mining application solves the online clustering problem. The program outputs a file containing the cluster centers found for the input data set. The output abstraction extracts the quality of the clustering as measured by the BCubed metric [4]. The sensitivity metric is the absolute difference between the results of the clustering metric. The resulting accuracy specification directly captures the ability of the application to solve the clustering problem.

The native PARSEC input consists of a randomly generated set of points drawn from a uniform distribution. Since for this input all clusterings have equal low quality clustering where even the trivial algorithm (that randomly selects cluster centers) have almost identical accuracy, we augment the evaluation input set with more realistic inputs from the UCI Machine Learning Repository [8].

Other Benchmarks. The PARSEC benchmark suite also contains the following benchmarks: facesim, dedup, fluidanimate, freqmine, and vips. We did not include freqmine and vips because these benchmarks did not successfully compile with the LLVM compiler version 2.5. We did not include dedup and fluidanimate because these applications produce complex binary output files. Not having deciphered the meaning of these files, we were unable to develop meaningful accuracy specification. We do not include facesim because its discards the output and produces only timing information.

2.3 Quantitative Exploration Results

We next present the results of applying SpeedPress to our benchmark applications with representative inputs and the accuracy specification presented in Section 2.2. The goal of this exploration is to find the upper bound to applicability of loop perforation and identify common properties of approximate computations that make loop perforation successful, and generalize the properties of these transformations beyond this specific transformation.

2.3.1 Sensitivity Profiling Results

We first execute the sensitivity profiling algorithm to find loops that can be perforated in each application. We collect the count of loops that the sensitivity profiling filtered out for different reasons. We also collect the number of loops that the testing procedure identified

as perforatable, which indicates the number of approximate computations in the benchmark applications.

Methodology. We used the benchmarks and inputs from Section 2.2. To start sensitivity profiling, we considered the loops (identified by profiling) that contribute at least 1% of the executed instructions (with a cutoff after the top 25 loops). We specified a sensitivity bound b of 0.1 (representing 10%). We instructed SpeedPress to perforate loops with interleaving perforation, which skips every other iteration. We applied four different perforation rates – 0.25 (skip a quarter of iterations), 0.5 (skip a half of iterations), 0.75 (skip three quarters of iterations), and execute a single loop iteration (skip all iterations after the first). We performed all of our runs on eight dual quad Intel Xenon X5460 3.1 GHz machines with 8 GB of RAM running Linux.

Profiling Results. Table 2.2 summarizes, for each application, the result of checking each loop in the sensitivity profiling algorithm from Section 2.1.3. Each column presents results for a given perforation rate (0.25, 0.5, 0.75 and 1 iteration). The first row (Candidate) presents the starting number of candidate loops. This number is always 25 unless the application has fewer than 25 loops that account for 1% of the executed instructions.

The second row (Crash) presents the number of loops that Algorithm 2.1 filters out because perforating the loop caused the application to crash or otherwise terminate with an error. The third row (Accuracy) presents the number of loops filtered by the algorithm because perforating the loop caused the application to violate the corresponding accuracy bound. The fourth row (Speed) presents the number of remaining loops that Algorithm 2.1 filters out because perforating the loop does not improve the overall performance (this typically happens for tight loops at a 0.25 perforation rate). The fifth row (Valgrind) presents the number of remaining loops that Algorithm 2.1 filters out because their perforation introduces a latent memory error detected by the Valgrind memcheck tool [86].

SpeedPress was able to find at least one perforatable loop in each of the benchmarks (these loops improve the program's performance, while satisfying the developer's accuracy requirement). The maximum number of perforatable loops is 13 (bodytrack, while running a single iteration of the loop). These results show that most of the benchmarks have only a few perforatable loops, which indicates that the applications have a small number of *approximate computational kernels*, comprising these perforatable loops. Modifying the approximate computational kernels affects only the accuracy of the computation, and not its proper execution (on the set of representative inputs). In Sections 2.4 and 2.5, we will focus on the function and structure of the approximate computations.

x264				
Filter	25%	50%	75%	1 iter
Candidate	25	25	25	25
Crash	1	1	1	1
Accuracy	6	7	7	6
Speed	16	12	10	11
Valgrind	0	0	1	1
Remaining	2	6	6	6

ferret				
Filter	25%	50%	75%	1 iter
Candidate	25	25	25	25
Crash	8	12	12	6
Accuracy	13	11	11	17
Speed	0	0	0	0
Valgrind	0	0	0	0
Remaining	4	2	2	2

bodytrack				
Filter	25%	50%	75%	1 iter
Candidate	25	25	25	25
Crash	2	5	7	1
Accuracy	1	1	2	2
Speed	12	10	1	1
Valgrind	3	1	6	8
Remaining	7	8	9	13

canneal				
Filter	25%	50%	75%	1 iter
Candidate	16	16	16	16
Crash	7	10	10	6
Accuracy	1	1	1	4
Speed	7	4	4	5
Valgrind	0	0	0	0
Remaining	1	1	1	1

swaptions				
Filter	25%	50%	75%	1 iter
Candidate	25	25	25	25
Crash	3	6	8	0
Accuracy	13	12	13	16
Speed	5	4	2	2
Valgrind	2	2	1	5
Remaining	2	1	1	2

blackscholes				
Filter	25%	50%	75%	1 iter
Candidate	6	6	6	6
Crash	0	0	0	0
Accuracy	4	4	4	4
Speed	1	1	1	1
Valgrind	0	0	0	0
Remaining	1	1	1	1

streamcluster				
Filter	25%	50%	75%	1 iter
Candidate	15	15	15	15
Crash	1	1	2	4
Accuracy	0	0	0	0
Speed	13	11	8	7
Valgrind	0	0	0	0
Remaining	1	3	5	4

Table 2.2: Sensitivity Profiling Results for Individual Loops. For Each Perforation Rate, Table Contains the Number of Loops Identified by Each of the Filters.

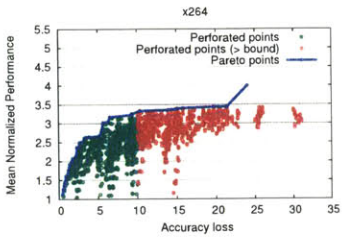


Figure 2.2: x264

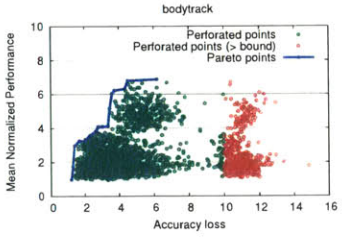


Figure 2.3: Bodytrack

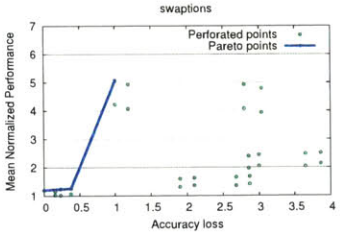


Figure 2.4: Swaptions

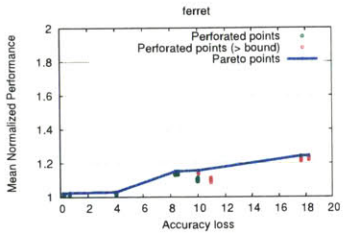


Figure 2.5: Ferret

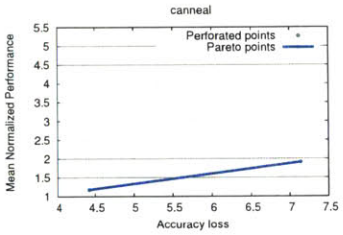


Figure 2.6: Canneal

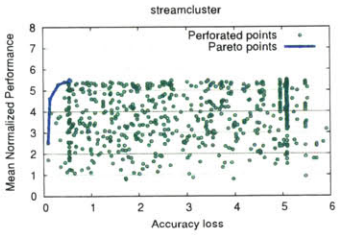


Figure 2.7: Streamcluster

2.3.2 Tradeoff Space Exploration Results

We execute SpeedPress's exhaustive loop perforation space exploration algorithm to get a sense for the approximation potential of the benchmark applications (i.e., the maximum amount of perforation that we can apply). For the set of programs transformed using loop perforation, SpeedPress identifies Pareto-optimal perforated programs.

Methodology. We used the benchmarks and inputs from Section 2.2 and the results from sensitivity profiling. After the algorithm calculated the set of Pareto-optimal perforated programs, we executed those that satisfy the sensitivity bounds of 2.5%, 5%, 7.5% and 10.0% on a different set inputs (also described in Section 2.2). We compared the obtained performance and accuracy results to check whether the results generalize to other inputs. We performed all of our runs on the same machines as sensitivity profiling.

Exploration Results. Figures 2.2 through 2.7 present the results of this exploration. The graphs plot a single point for each explored perforation. The y coordinate of the point is the mean speedup of the perforation (over all profiling inputs). The x coordinate is the corresponding percentage accuracy loss of the perforation. Green points have accuracy losses below 10%; red points have accuracy losses above 10%. The blue line in each graph connects the points from Pareto-optimal perforations.

Training Results

Application	Bound			
	2.5%	5%	7.5%	10%
x264	2.38 (2.5%)	2.66 (5%)	3.17 (6.53%)	3.25 (9.31%)
bodytrack	3.44 (2.23%)	6.32 (4.36%)	6.89 (6.19%)	6.89 (6.19%)
swaptions	5.08 (1.0%)	5.08 (1.0%)	5.08 (1.0%)	5.08 (1.0%)
ferret	1.02 (0.2%)	1.03 (4%)	1.03 (4.0%)	1.16 (10.0%)
canneal	1.14 (4.38%)	1.18 (4.43%)	1.91 (7.14%)	1.91 (7.14%)
blackscholes	33.0 (0.0%)	33.0 (0.0%)	33.0 (0.0%)	33.0 (0.0%)
streamcluster	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)	5.51 (0.54%)

Production Results

Application	Bound			
	2.5%	5%	7.5%	10%
x264	2.34 (5.15%)	2.53 (6.08%)	3.12 (8.72%)	3.19 (10.3%)
bodytrack	2.70 (4.00%)	4.93 (6.12%)	4.81 (6.58%)	4.81 (6.58%)
swaptions	5.05 (0.20%)	5.05 (0.20%)	5.05 (0.20%)	5.05 (0.20%)
ferret	1.00 (0.15%)	1.02 (0.23%)	1.02 (0.23%)	1.07 (7.90%)
canneal	1.14 (4.38%)	1.14 (4.38%)	1.46 (7.88%)	1.46 (7.88%)
blackscholes	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)	28.9 (0.0%)
streamcluster	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)	4.87 (1.71%)

Table 2.3: Training and Production Results for Pareto-optimal Perforations for Varying Accuracy Bounds. A pair x (y%) presents the corresponding mean speedup x and mean accuracy y

The graphs show that, for these applications, loop perforation is usually able to increase performance on the profiling inputs by at least a factor of two (up to a factor of seven for bodytrack) while reducing the accuracy by less than 10% (ferret is the exception). The graphs also illustrate the broad range of points in the performance vs. accuracy trade-off space that the loop perforation space exploration algorithm is able to find. Overall, this indicates that the benchmark applications have a substantial optimization potential that can be uncovered by perforating individual loops and combinations of loops.

Sensitivity of Tradeoffs to Different Inputs. Table 2.3 presents accuracy and speedup results for selected Pareto-optimal perforations in the loop perforation space. There is a row for each application and a group of columns for the profiling and production inputs. Each group of columns presents results for the Pareto-optimal perforation for four accuracy loss bounds b: 2.5%, 5%, 7.5%, and 10%. Each entry of the form $X(Y\%)$ presents the corresponding mean speedup X and mean accuracy Y for that combination of application,

Application	Sensitivity Profiling (Individual Loops)			Tradeoff Exploration
	Accuracy	Valgrind	Total	(Multiple Loops)
x264	500 (108m)	110 (840m)	610 (949m)	3071 (665m)
bodytrack	100 (35m)	47 (1316m)	147 (1351m)	5624 (1968m)
swaptions	100 (7m)	16 (108m)	116 (115m)	32 (9m)
ferret	100 (17m)	40 (53m)	140 (71m)	255 (43m)
canneal	256 (405m)	60 (540m)	316 (945m)	2 (12m)
blackscholes	24 (0.5m)	12 (5m)	36 (5.5m)	19 (1m)
streamcluster	500 (3083m)	17 (782m)	517 (3865m)	639 (3941m)

Table 2.4: Sensitivity Profiling Statistics for Benchmark Applications.

bound, and input set. With the exception of ferret, all applications show a reasonable correlation between profiling and production results, indicating that the results from the sensitivity profiling generalize well to other inputs.

2.3.3 Execution Time of Analysis Results

Table 2.4 presents timing results for the sensitivity profiling runs (Algorithm 2.1). The table contains a row for each application. The second column Accuracy presents timing results for executions that measure the speedup and accuracy of different perforations. The third column (Valgrind) presents timing results for the Valgrind dynamic analysis that searches for memory errors. The fourth column (Total) presents the sum of results from columns two and three. The fifth column (Tradeoff Exploration) presents timing results for exploring the combinations of perforated loops. Each entry of the form $X(Y)$ indicates that algorithm considered X different combinations of perforated loops and that the executions took a total of Y minutes to complete.

The total execution times for sensitivity profiling range from 6 minutes for blackscholes to approximately 64 hours for streamcluster, with other applications requiring significantly less time. The total execution times range from 1 minute for blackscholes to 65 hours for streamcluster. These execution times include both the execution of the program and the additional sensitivity profiling of the combined perforations with Valgrind.

2.4 Computational Patterns Amenable to Loop Perforation

This section presents the properties of approximate computations. We identified these properties by examining perforatable loops in the benchmark applications (discovered by the quantitative exploration in Section 2.3). Then in Section 2.5 we present the computations we identified in the benchmark applications that are instances of these patterns. Based on our investigation, we present two kinds of patterns:

- **Functional patterns.** These patterns focus on how the rest of the application uses the approximate computations by describing how the results of the computation contribute to computing the program's output. For instance, a computation may calculate a distance metric that the application uses to compare the fitness of multiple elements before returning the fittest element; or a computation may perform Monte-Carlo simulation that computes one of the application's results.
- **Structural patterns.** These patterns describe the statements and expressions that comprise the pattern and specify the points in the code that the transformation can modify. For example, a computation may represent a sum that aggregates the values computed in each loop iteration; in this loop, perforation can change the expression that increments the induction variable and the expression that checks the loop's exit condition.

We remark that while functional and structural patterns may often apply together to the same computation, they are conceptually different. For instance, a distance metric is often computing as an average of element-wise differences. However, a Monte-Carlo simulation may also compute the average of the large number of individual trials. While the structure of the computation in this case is the same – and we can *analyze* these computations using the same analysis – the role of the computation can clearly be different – and therefore we can have different goals when *specifying the accuracy requirements* of the computation.

Methodology. Our analysis of approximate computation focuses on those computations that are amenable to loop perforation. We specifically studied loops that appear in Pareto-optimal perforations (Section 2.3.2), as those loops contribute most to the approximate program's accuracy and performance. To identify likely patterns, we investigated the source code of the application and used debugging tools to understand the behavior of applications. In almost all cases, we defined patterns after identifying the common properties in two or more studied computations.

2.4.1 Functional Patterns

In this section we describe the *functional computational patterns* that interact well with loop perforation. We identified the following patterns (see Section 2.5 for a discussion of the pattern instances, which we identified in the benchmarks):

Search Space Enumeration (SSE): The application iterates over a search space of items. The perforated computation skips some of the items, returning one of the items from the remaining part of the search space.

Search Metric: The application uses a *search metric* to drive a search for the most desirable item from a set of items:

- **Selection (SMS):** A *selection* metric quantifies the desirability of each item encountered during the search.
- **Filtering (SMF):** A *filtering* metric determines if the search should remove the item from a set of active items.
- **Termination (SMT):** A *termination* metric determines if the search should terminate, either because an acceptable item has been found or because the likelihood of finding a more desirable item appears to be small.

Perforating a search metric computation produces a new, less accurate but more efficiently computable metric. The effect is that the perforated search may return a less desirable but still acceptable item. We note that some applications use the same metric for more than one purpose, i.e., a metric can be a combined selection, filtering, and/or termination metric.

Monte-Carlo Simulation (MC): The application performs a Monte-Carlo simulation. The perforated computation evaluates fewer samples to produce a (potentially) less accurate result more efficiently.

Iterative Improvement (II): The application repeatedly improves an approximate result to obtain a more accurate result. The perforated computation performs fewer improvement steps to produce a (potentially) less accurate result more efficiently.

Array Update (UPD): The application traverses an array, updating its elements. The perforated computation skips some of the updates, leaving the previous values in place.

Fully Redundant Computation (RED): This application produces data that is not later used. The perforated computation skips redundant computation.

2.4.2 Structural Patterns

We have identified several structural computational patterns that interact well with loop perforation. Figure 2.8 presents four patterns that we identified in the perforated computations. All four patterns perform *reduction* – they aggregate data computed in each iteration and produce a single result.

In each pattern, the computation inside the loop body is denoted as a generic function `f()`. While it can read the induction variable `i` and other state variables surrounding the loop, the function is side-effect free. It only produces the value and does not otherwise affect the program state. The type of variables is also generic, denoted as `val`, except for the type of the induction variable, which is always an integer (`int`).

Sum Pattern

```
val sum = 0.0;
for (int i = 1; i <= n; i++) {
  sum += f(i);
}
return sum;
```

Average Pattern

```
val sum = 0.0;
for (int i = 1; i <= n; i++) {
  sum += f(i);
}
val mean = sum / n;
return mean;
```

Argmin Pattern

```
val best = MAX_DOUBLE;
int best_index = -1;
for (int i = 1; i <= n; i++) {
  val s = f(i);
  if (s < best) {
     best = s; best_index = i;
  }
}
return best_index;
```

Ratio Pattern

```
val numer = 0.0;
val denom = 0.0;
for (int i = 1; i <= n; i++) {
  numer += f1(i);
  denom += f2(i);
}
ratio = numer / denom;
return ratio;
```

Figure 2.8: Local Perforation Patterns. Loop Perforation Transforms Shaded Code Blocks.

Sum Pattern. The loop sums up the values computed in each iteration. In each iteration, the computation inside the loop body, represented by the function `f()`, produces a single contribution.

Average Pattern. The loop sums up the values computed in each iteration and computes the average by dividing the running sum with the number of elements.

Argmin (Argmax) Pattern. This pattern compares the value computed in each loop iteration against the minimum (maximum) value that was observed so far. The computation saves and returns the iteration count of the minimum computed element. This pattern often represents computations that iterate over collections.

Ratio Pattern. This pattern consists of a single loop that computes two aggregate values, numerator (`numer`) and denominator (`denom`) using the function `f1` and `f2`, respectively. After the loop, the computation computes the ratio of the two sums. One can see this pattern as the generalization of the average pattern.

2.5 Analysis of Perforated Computations in Benchmarks

Table 2.5 presents data for every loop which is perforated in at least one Pareto-optimal approximate program with an accuracy loss under 10%. The first column presents the function in which the loop appears and (when the loop appears in a loop nest) whether the loop is an inner loop or outer loop of a loop nest. The second column presents the percentage of time spent in each loop (before perforation). The third column presents both the functional (as discussed in Section 2.4.1) and structural (as discussed in Section 2.4.2) computation patterns for the loop. For example, the entry for the first loop in x264 is SSE/Argmin, which indicates that the functional pattern for the loop is the Search Space Enumeration pattern, while the structural pattern is the Argmin pattern. Some loops are not an instance of any of the structural patterns. In this case we present the functional pattern only.

2.5.1 x264

The x264 encoder divides each frame into *blocks*, then performs the encoding at the granularity of blocks. *Motion estimation* attempts to find a block from a previously encoded frame that is similar to the block that x264 is currently attempting to encode. If it finds such a block, x264 encodes the delta over this previously encoded block. Motion estimation consumes the vast majority of the computation time in x264.

All of the perforatable loops in x264 appear in the motion estimation computation. Two of these loops (`x264_mb_analyze_inter_p16x16` and `x264_search_ref`) are instances of the Search Space Enumeration functional pattern and the Argmin structural pattern (which computes

x264		
Function	Time	Pattern Type
x264_mb_analyse_inter_p16x16	64.20%	SSE / Argmin
x264_pixel_sad_16x16, outer	55.80%	SMS+T / Sum
x264_pixel_sad_16x16, inner	54.60%	SMS+T / Sum
x264_me_search_ref	25.00%	SSE / Argmin
pixel_satd_wxh, outer	18.50%	SMS+T / Sum
pixel_satd_wxh, inner	18.30%	SMS+T / Sum
bodytrack		
Function	Time	Pattern Type
Update	77.00%	II
ImageErrorInside, inner	37.00%	SME / Ratio
ImageErrorEdge, inner	29.10%	SME / Ratio
InsideError, outer	28.90%	SME / Sum
IntersectingCylinders	1.16%	SMF+SSE
swaptions		
Function	Time	Pattern Type
HJM_Swaption_Blocking, outer	100.00%	MC / Mean
HJM_SimPath_Forward_Blocking, outer	45.80%	RED
HJM_SimPath_Forward_Blocking, inner	31.00%	RED
Discount_Factors_Blocking	1.97%	UPD
ferret		
Function	Time	Pattern Type
emd	37.60%	SMS+II
LSH_query_bootstrap	27.10%	SSE
LSH_query_bootstrap	26.70%	SSE
LSH_query_bootstrap	2.70%	SSE
canneal		
Function	Time	Pattern Type
reload	2.38%	UPD
blackscholes		
Function	Time	Pattern Type
bs_thread	98.70%	RED
streamcluster		
Function	Time	Pattern Type
pFL, inner	98.50%	II
pgain	84.00%	SME+T+UPD
dist	69.30%	SME+T / Sum
pgain	5.01%	SME+T

Table 2.5: Patterns in Pareto-optimal Perforations. Pattern Type lists loop's functional and (optional) structural patterns. These patterns are described in Sections 2.4.1 and 2.4.2, respectively.

the index and value of the minimum element in an array of elements). The most time consuming loop (`x264_mb_analyze_inter_p16x16`) searches previously encoded reference frames to find a block that is a good match for the block that x264 is currently encoding. Perforating this loop causes x264 to search fewer reference frames. The second (`x264_search_ref`) is given a single reference frame and searches within that frame to find a good match. Perforating this loop causes x264 to consider fewer blocks within the frame.

All of the remaining loops are instances of the Search Metric for both Selection and Termination functional pattern and the Sum structural pattern (which computes the sum of a set of numbers). These loops all compute a metric that measures how well the current block matches a previously encoded block. Perforating these loops causes x264 to skip pixels when it computes the metric, producing a new, less accurate, but more efficiently computable metric. In addition to using this metric to select the previously encoded block that is the best match, x264 also uses the metric to decide when it should terminate the search (within a frame, x264 terminates the search when it fails to find a new desirable block after a given number of probes). Because the perforated metric makes fewer distinctions between blocks, perforation will typically cause the search to terminate with fewer probes.

All of these perforations may cause x264 to choose a less desirable previously encoded block as a starting point for encoding the current block. But because there is significant redundancy in the set of previously encoded blocks (typically, many previously encoded blocks are a reasonably good match for the current block), the results show that x264 is still usually able to find a good block even after perforation.

2.5.2 Bodytrack

Bodytrack uses an annealed particle filter to track the movement of specific parts of a human body (head, torso, arms, and legs) as a person moves through a scene [35]. At each input video frame bodytrack starts with a probabilistic model of the position of the body from the previous frame. This model consists of a weighted set of *particles*. Each particle consists of a set of angles between body parts that together specify a body pose. Each particle is assigned a weight (a likelihood that it accurately represents the current body pose). The goal of the computation is to calculate a new model of the body in the current frame as a weighted average of the particles. Bodytrack starts with the model from the previous frame, then refines the model by iterating through multiple annealing layers. At each annealing layer it processes each particle to create a representation of the body position and location as a set of cylinders, each of which represents a specific body part. It then compares this

representation to image processing data from the current frame, then uses the comparison to update the weight of the corresponding particle.

The `Update` loop performs the annealing steps. Because these steps are designed to improve the accuracy of the model, we identify this loop as an instance of the Iterative Improvement functional pattern (even though it is possible for an individual step to produce a slightly less accurate model). Perforating this loop causes bodytrack to perform fewer annealing steps and produce a potentially less accurate model more efficiently.

The `ImageErrorInside`, `ImageErrorEdge`, and `InsideError` loops compute a metric that characterizes how closely the body pose for a given particle matches the observed image data. These loops select a number of sample points along the edges and interiors of the cylinders that represent the body position. We identify these loops as instances of the Search Metric for Selection pattern — they compute values that measure how closely the particle matches the image data. Two of these loops are instances of the Ratio structural pattern (which computes the ratio of two sums). The third is an instance of the Sum structural pattern. Perforating these loops causes bodytrack to consider only a subset of the sample points when it computes the metric. The result is a more efficiently computable but potentially less accurate metric.

The `IntersectingCylinders` loop iterates over all pairs of cylinders in a given particle to compute if any of the pairs intersect. If so, bodytrack removes the particle from the model (and may potentially replace it with a new particle). We identify this loop as an instance of the Search Metric for Filtering pattern because it computes a simple metric (either the particle is valid or invalid). We also identify this loop as an instance of the Search Space Enumeration pattern because it enumerates all pairs of cylinders. Perforating this loop causes bodytrack to consider only a subset of the pairs of cylinders. The result is that bodytrack may consider a particle to be valid even though it represents an unrealistic body pose. Bodytrack will therefore keep the particle in the model instead of filtering it out. Because these particles represent unrealistic positions, they will typically be given little or no weight in the model and will therefore have little or no effect on the accuracy. Note that this effect may actually decrease the performance (although we observed little or no impact on the performance or accuracy in practice).

2.5.3 Swaptions

Swaptions uses Monte Carlo simulation to price a portfolio of swaptions. Each iteration of the `HJM_Swaption_Blocking` loop computes a single Monte-Carlo sample. Each complete execution of this loop computes the price of a single swaption. Perforating this loop causes

swaptions to compute each swaption price with fewer Monte-Carlo samples. Because of the way the computation is coded, the perforation produces prices that are predictably biased by the perforation rate. The system therefore uses extrapolation (as described in [95]) to correct the bias.

The `HJM_SimPath_Forward_Blocking` loop updates the matrix representing the path of the forward interest rate. While this matrix is later used to calculate the swaption price, the computation does not access the majority of elements after the second time step in the future (i.e., the second row of the table). Perforating the computation leaves the skipped matrix elements at their initial value, while avoiding the computation of unnecessary values. The `Discount_Factors_Blocking` loop updates a data structure containing discount factors used to compute the price of the swaptions. Perforating the computation leaves the skipped elements at their initial value of 1 instead of the original initial value. Our results show that both of these perforations have a small impact on the accuracy of the computation.

2.5.4 Ferret

Given a query image, ferret performs a content-based similarity search to return the N images in its image database most similar to the query image. Ferret first decomposes the query image into a set of segments, extracts a feature vector from each segment, indexes its image database to find candidate similar images, ranks the candidate images by measuring the distance between the query image and the candidate images, then returns the highest ranked images. The computation has two main phases. The first phase uses an efficient hash-based technique to retrieve a fixed-length (double the number of requested images) list of likely candidate images from the database. The second phase performs a more accurate comparison between the retrieved images and the query image.

The `LSH_query_bootstrap` loops execute as part of the first phase. The first two loops iterate over lists of images indexed in selected hash buckets to extract the set of candidate images from the database. Perforating these loops causes ferret to skip some of these images so that they are not considered during the subsequent search (these images will therefore not appear in the search result). We identify these loops as instances of the Search Space Enumeration pattern because they iterate over (part of) the search space.

The remaining `LSH_query_bootstrap` loop finds where to insert the current candidate image in the fixed-length sorted list of search results. Perforating this loop may cause the candidate image to appear in the list out of order. Because the second phase further processes the images in this list, the final set of retrieved images is presented to the user in sorted order.

The `emd` loop executes as part of the second phase. This loop computes the *earth mover* distance metric between the query image and the current candidate image from the image database. Because ferret uses this metric to select the most desirable images to return, we identify the loop as an instance of the Search Metric for Selection pattern. This metric is also used for the final sorting of the images according to desirability. Interestingly enough, this search metric is implemented as an instance of the Iterative Improvement pattern – it improves the distance estimate until it obtains an optimal distance measure or exceeds a maximum number of iterations. Perforating this loop creates a new, more efficient, but less accurate metric. As a result, the program may return a different set and/or ordering of images to the user.

2.5.5 Canneal

Canneal uses simulated annealing to place and route an input netlist with the goal of minimizing the total wire length. The reload loop traverses the state vector for the canneal Mersenne twister random number generator to reinitialize the values in this vector. For our set of inputs, the resulting change in the generated sequence of random numbers causes canneal to execute marginally more efficiently.

2.5.6 Blackscholes

The experimental results show that it is possible to perforate the outer loop in `bs_thread` without changing the output at all. Further investigation reveals that this loop simply repeats the same computation multiple times and was apparently added to the benchmark to increase the workload.

2.5.7 Streamcluster

Streamcluster partitions sets of points into clusters, with each cluster centered around one of the points. Each clustering consists of a set of points representing the cluster centers. The goal is find a set of cluster centers that minimizes the inter-cluster and intra-cluster distances. Streamcluster uses a version of the *facility location* algorithm to find an approximately optimal clustering. The facility location algorithm contains a while loop that executes a sequence of clustering rounds, each of which attempts to improve the clustering from the previous round. The while loop terminates if a round fails to generate a clustering with significantly improved cost.

The pFL loop executes once per round. Each iteration of this loop generates (by adding a randomly chosen new candidate cluster center to the current set of cluster centers) and evaluates a new candidate clustering. If this candidate clustering improves on the current clustering, the loop body updates the current clustering (optionally merging some of the clusters). We identify this loop as an instance of both the Search Space Enumeration pattern (because it iterates over a part of the search space of candidate clusterings) and the Iterative Improvement pattern (because it uses the current clustering to generate improved clusterings). Perforating the pFL loop therefore causes streamcluster to consider fewer candidate clusterings per round. The result is that streamcluster performs fewer attempts to improve the clustering before the next round termination check, which may in turn cause streamcluster to produce a less desirable clustering more efficiently.

We note that the following comment appears in the source code above the definition of the constant (ITER) that controls the number of iterations of the pFL loop:

```
/* higher ITER --> more likely to get correct number of centers */
/* higher ITER also scales the running time almost linearly */
```

This comment reflects the fact that the number of iterations of the pFL loop controls a performance vs. accuracy tradeoff (which is not exposed to the user of the application). In effect, the perforation space exploration algorithm rediscovers this tradeoff.

The first pgain loop calculates the partial cost of a candidate clustering by computing sums of distances between data points and the new cluster center. It also marks the data points that would be assigned to the new cluster center. The second pgain loop uses the computed partial sums to compute the total cost of the candidate clustering. We identify these loops as instances of the Search Metrics for Selection (because the computed costs are used to select desirable clusterings) and Termination (because the facility location algorithm uses this cost as a measure of progress, and stops if the progress is too small) pattern. Perforating these loops produces a less accurate but more efficiently computable cluster cost metric. We also identify the first pgain loop as an instance of the Data Structure Update pattern. Perforating this loop may leave some of the data points assigned to an old cluster, even though these points should be assigned to the newly opened cluster.

The dist loop computes the distance between two points. We identify this loop as an instance of the Search Metric for Selection and Termination pattern because it is used to compute candidate clustering costs. It is also an instance of the Sum structural pattern. Perforating this loop causes streamcluster to compute the distance between points using a subset of the coordinates of the points. The result is a more efficiently computable but less accurate distance metric.

Application	Computation	Location	Exec. Time %	Mean Iterations	Loop Runs
bodytrack	`ImageErrorInside`	`ImageMeasurements.cpp`, 142	37.0%	40	202824
swaptions	`HJM_Swaption_Blocking`	`HJM_Swaption_Blocking.cpp`, 157	99.9%	1250	64
streamcluster	`pFL`	`streamcluster.cpp`, 600	98.5%	52	49
x264	`x264_search_ref`	`me.c`, 411	25.0%	15.5	3581705

Table 2.6: Execution Statistics for Example Structural Pattern Computations

2.6 Analysis of Perforated Kernel's Absolute Error

The previous section described the function and structure of the approximate computations that interact well with loop perforation. This section investigates the numerical error of individual perforated computations. We specifically focus on computations that are instances of the structural computational patterns and study the nature of errors that perforation introduces compared to the original version of the computation.

Methodology. We selected four computations that belong to each of the four structural patterns. Specifically, we analyzed four representative computations that appear in Pareto-optimal perforated programs (see Table 2.5) that consume a significant amount of the benchmark's execution time. Table 2.6 presents the execution statistics for the perforated loops. The first two columns present the application and computation names. The third column (Location) presents the file name and the line where the loop begins. The fourth column (Execution Time %) presents the percentage of instructions executed within the computation. The fifth column (Mean Iterations) presents the mean number of iterations that the loop executes. Finally, the sixth column (Loop Runs) presents the number of times the loop was executed. This number corresponds to the number of the outputs that the computation produces during the lifetime of the application. We use these parameters to compute the worst-case error for the computations.

We perforate each subcomputation using the sampling perforation strategy with three perforation rates, $r \in \{0.25, 0.50, 0.75\}$. For each subcomputation we record the inputs and *local error* of the perforated version of the computation. The local error is an absolute error between the result of the original and perforated subcomputations. It captures only the error that emerges in a single execution of the perforated computation. It does not include the error that accumulated from possible previous perforated executions.

To compute the local error, our manual instrumentation copies and executes side-by-side the original and perforated versions of the computation. Conceptually, if the original com-

putation is `f()` and perforated is `f'()`, then the instrumented computation has the form `r = f(); r' = f'(); record (r, r'); return r;` We ensure that the perforated computation does not have side-effect or otherwise change the original execution. Therefore, the execution continues by using the result `r` produced by the original program execution.

We used selected representative inputs from Section 2.2. For bodytrack, we used the first 20 frames of the `sequenceA` input provided by benchmark developers. For swaptions, we used the `simlarge` input that comes with the benchmark suite. For streamcluster, we used 10^5 points from the `animalNorm` dataset from UCI Machine Learning Repository [8] For x264, we used a sequence of 60 frames from the `tractor` video sequence from Xiph.org Foundation web site.

2.6.1 Worst-Case Absolute Error Analysis

To compare the errors that the computations experienced and the maximum errors that the computations *may* have, we derived the expressions for the worst-case error for the structural patterns from Figure 2.8 as functions of the the percent of skipped iterations, r:

Sum Pattern. The analysis assumes that the result of the function `f()` is in the range $[a, b]$, and the perforated computation executes only $\lfloor r \cdot n \rfloor$ iterations. Then, the worst-case error is $(n - \lfloor r \cdot n \rfloor) \cdot (b - a)$.

Average Pattern. The analysis assumes that the result of the function `f()` is in the range $[a, b]$. When the perforated computation executes $\lfloor r \cdot n \rfloor$ iterations, the worst-case error is $(1 - \frac{1}{n} \cdot \lfloor r \cdot n \rfloor) \cdot (b - a)$.

Argmin (Argmax) Pattern. The analysis assumes that the the result of the function `f()` is in the range $[a, b]$. To express the error of the computation, we specify that the error selecting an inexact index is proportional to the difference of the computed value `best` in perforated execution from its value in the exact execution.

Even skipping a single iteration of the loop incurs the error $b-a$. This error corresponds to the case when the skipped iteration would produce the minimum value a and all remaining `L-1` iterations produced the value b. This analysis demonstrates that the worst-case error of the minimum (and maximum) function, unlike the analysis of the sum, is more sensitive to perforation – the error is either 0 when perforation is not applied ($r = 0$), or maximum error when the perforation is applied ($r \neq 0$).

Ratio Pattern. We consider the case when the result of the function `f1()` is in the range $[a, b]$ and the result of the function `f2()` is in the range $[c, d]$.

If the range $[c, d]$ contains the value 0 (i.e., c and d have the different signs), then the worst case error is infinite, since the perforation may cause a division by zero. Otherwise, if c and d have the same sign, then we analyze the error as follows. Let $s_1 = \sum_1^n \texttt{f1}_i$ be the numerator, $s_2 = \sum_1^n \texttt{f2}_i$ be the denominator. If $n' = \lfloor r \cdot n \rfloor$, then the perforated loop computes the perforated numerator and denominator $\hat{s}_1 = \sum_1^{n'} \texttt{f1}_i$ and $\hat{s}_2 = \sum_1^{n'} \texttt{f2}_i$, respectively. We express the error as $|(s_1 \cdot \hat{s}_2 - \hat{s}_1 \cdot s_2)/(s_2 \cdot \hat{s}_2)|$. This error reaches the maximum when the difference in its numerator reaches maximum and the product in its denominator reaches minimum, and is equal to $\frac{n-n'}{n} \cdot \frac{|b \cdot d - a \cdot c|}{\min(c^2, d^2)}$.

2.6.2 Error Analysis Results

Table 2.7 presents the observed and the worst-case bounds on the absolute error. The first and the second columns present the name of the application and the analyzed pattern. The third column (Observed Input Range) represents the range of inputs collected from the program's execution. The fourth column (Perforation Rate) presents the amount of skipped loop iterations. The columns five to seven present the local error observed while running the representative inputs. Column 5 presents the local error that is 95th percentile (i.e., it is greater than 95% of the observed errors). Column 6 presents the 99th percentile of the local error and Column 7 presents the maximum observed local error. Finally, Column 8 presents the worst-case error computed using the expressions in Section 2.6.1 and the maximum observed input ranges.

The comparison of the worst-case and the observed error shows that the observed errors are significantly smaller than the worst-case error. In particular, the worst-case error is between 1.3 times (bodytrack, $r = 0.25$) and 576 times (swaptions, $r = 0.25$) greater than the maximum observed error, even though the worst-case error was calculated on the *observed* input intervals and the worst-case error analyses for the patterns are tight – i.e., it is possible to observe the intervals produced by the analysis.

The comparison of the worst-case error with the 95th and 99th percentile errors (i.e., errors that are greater than 95% and 99% of the observed errors, respectively) shows that the relatively small number of errors is in the tail of the distribution. For instance, the worst-case error is between 217 times (swaptions, $r = 0.25$) and 3.5 times (bodytrack, $r = 0.50$) greater than the 95th percentile error. Moreover, the argmin computation in x264 benchmark often produces the correct result even when perforated. For perforation

Application	Pattern	Observed Input Range	Perforation Rate	Observed Error			Worst-Case Error
				$\cdot > 95\%$	$\cdot > 99\%$	Max.	
bodytrack	ratio	$[a,b] = [77, 311]$ $[c,d] = [1145, 1223]$	0.25	0.016	0.022	0.044	0.056
			0.50	0.032	0.040	0.068	0.111
			0.75	0.037	0.047	0.112	0.168
swaptions	mean	$[a,b] = [0.0, 0.2]$	0.25	0.00023	0.00024	0.00024	0.05
			0.50	0.00015	0.00015	0.00015	0.10
			0.75	0.00022	0.00026	0.00026	0.15
streamcluster	sum	$[a,b] = [0, 12371]$	0.25	1144	18805	18805	160823
			0.50	9542	18805	91252	309275
			0.75	18805	19623	28698	457727
x264	argmin	$[a,b] = [2, 9413]$	0.25	0.00	0.00	1903	9411
			0.50	0.00	0.00	4081	9411
			0.75	0.00	13.0	4081	9411

Table 2.7: Observed and the Worst-Case Local Error of Perforated Computations.

rates 0.25 and 0.50, more than 99% of the approximate executions do not experience *any* error – i.e., the perforated computation was able to find the minimum element.

A comparison between the observed errors across different perforation rates (0.25, 0.50, and 0.75) shows that in all benchmark the magnitude of errors for the same frequency of large errors (95% or 99%) increases with the perforation rate of the computation. This local error increase is consistent with the increase of the global program error (Section 2.3.2), but the rate of error increase for these computations can be non-linear with the respect to the increase of perforation rate. For instance, the examination of the loop in streamcluster shows that each iteration computes the value that represents heuristic cost of opening a new cluster center. In most iterations this value is close to zero, but sometimes it is much larger. Therefore skipping some of the iterations that skip some of the larger values causes the greater variation in the magnitude of observed error.

2.7 Discussion

The experimental results presented in Section 2.3 show that loop perforation can effectively augment a range of applications with the ability to operate at various attractive points in the tradeoff space. The experimental results also present several key properties of the computations that can be approximated using loop perforation. This section discusses experimental results presented in this chapter and provides an insight that can help us build the formal foundation of the optimization with accuracy-aware transformations.

2.7.1 Approximate Kernels

Sensitivity profiling (Section 2.3.1) identified that the approximate benchmarks spend the majority of time in one or several loops that represent approximate kernel computations. Our manual examination of the kernels (Section 2.5) shows that each execution of kernel computations consumes a fraction of the application's input and contributes to producing a fraction of the program's output. Therefore, approximations that (like loop perforation) alter these kernels to perform less work, cause the program to produce less accurate parts of the output.

Tunable Approximate Knobs. In this chapter, we presented transforming kernel computations using loop perforation. A manual inspection of these computations shows that these computations are also amenable to other forms of approximation. Specifically, some of these computations have explicitly exposed parameters that control the manually implemented choice between the accuracy/performance tradeoff. For other kernels, such choice is hidden from the user, but the developers selected a fixed approximation choice (we presented a developer's comment for an approximation in streamcluster in Section 2.5.7). In our previous work, we also manually implemented alternative approximate implementations of these computations [79]. In all these cases, loop perforation opened new, or augmented existing ways to approximate the application.

Functional and Structural Patterns. We examined the kernel computations that are amenable to loop perforation and identified common functional and structural properties of these computations. In Section 2.4 we outlined these common properties as a set of computational patterns. While this set of patterns is not exhaustive, it covers a number of interesting cases that our evaluation with loop perforation uncovered.

These two classes of patterns are complementary. Structural patterns help us understand why the computation is amenable to approximation (e.g., loops that aggregate data are good candidates for loop perforation). Functional patterns help us understand how the application uses the result of the kernel execution to produce its result (e.g., each execution of perforated loops in x264 finds how to better compress the input data).

In addition to understanding the effect of loop perforation, identification of these patterns also guides the construction of a foundation for the rigorous analysis of approximate computations. First, we can exploit the structural patterns to define program analysis techniques that analyze the effects of the transformations. Second, we can exploit the functional patterns to provide a formal specification that the transformed computation should satisfy to fulfill the application's accuracy goal.

Of course, this set of patterns is not exhaustive. The evaluation shows that loop perforation is successful in identifying computations that aggregate data computed in loop's iterations. However, an example of a computation not amenable to loop perforation is one that in each iteration computes a single element value in an output array. Skipping iterations of such loops does not produce these array elements and results in unacceptable final output. For such computations, we discuss one alternative developer-guided approach to sensitivity profiling in Chapter 4.

Magnitude and Frequency of Errors. Section 2.6 presents the comparison of the worst-case error analysis and errors observed for several approximate perforated kernels. It shows that the approximate kernels rarely produce large errors – the typical errors were an order of magnitude smaller than the worst-case error (and the full program executions did not experience the maximum errors, even for a large number of kernel executions). These experimental results indicate that the worst-case analysis alone is insufficient to reason about the accuracy of the program.

To augment the understanding of the accuracy of transformed kernels, a more general property of interest is *how often the approximate computations exhibit large error.* An analysis that can answer this question reasons about the distribution of errors and can consequently analyze both the magnitude and frequency of acceptable errors. Such analysis generalizes both the worst-case analysis (which is the error that is greater than 100% of possible errors) and typical-case analyses (e.g., ensuring that with probability 0.95, the magnitude of error is smaller than a provided bound, such as the statistical analysis in 2.6).

2.7.2 Limitations of Testing-Based Accuracy-Aware Optimization

Our experimental results indicate that we can use testing-based techniques (that only use sensitivity profiling for accuracy-aware optimization) to optimize complex approximate programs, their dependence on developer-provided representative inputs limits their power in several ways.

Lack of Accuracy Guarantees. Our evaluation in Section 2.3 shows that for approximate benchmarks the programs optimized on one set of representative inputs often provide similarly accurate results for other inputs. However, in some cases the applications can experience visible differences when executed on a different set of inputs (e.g., in some configurations of ferret and x264 benchmarks). Overall, providing stronger guarantees about the accuracy would require a conceptually different approach that allows specifying a class of inputs (e.g., represented by input intervals or distributions) instead of individual inputs.

Lack of Safety Guarantees. While sensitivity profiling can identify errors that exist in concrete executions (and our examination in Sections 2.3.1 and 2.5 shows its effectiveness), it cannot provide guarantees about safety of transformed programs. Examples of concrete safety properties that the developer may be interested in include pointer safety and output integrity. For instance, a perforated loop should not leave unallocated array elements; a perforated argmin loop in x264 should always produce a legal index of the minimum element in the list. Likewise, a transformed distance computation in streamcluster should still satisfy the properties of a distance metric (i.e., non-negativity, coincidence, symmetry, and subadditivity). In our other work, we presented techniques for automatically checking several key safety properties [17]. Michael Carbin also presented a general interactive verification system for reasoning about arbitrarily complex safety and worst-case accuracy predicates in approximate computations [15, 16].

Scalability of Tradeoff Space Exploration. The execution time of the tradeoff space exploration algorithm is proportional to the number of combinations of transformations that we can apply and the number of representative inputs. Section 2.3 shows that such an approach is feasible for a transformation like loop perforation that is typically applied to a small number of loops. However, even for the case when we have combinations of 5 loops with different perforation rates, the search can exceed 60 hours (as in case of streamcluster). Such complexity can be overly prohibitive for transformations that are designed to be applied to program locations at a finer granularity than loops (e.g., approximation of arithmetic operations).

To address some of these limitations, the chapters that follow discuss techniques for accuracy analysis that provide guarantees on the frequency of large output deviations and algorithms for navigating tradeoff spaces equipped to optimize large space that are outside of the reach of the test-based techniques.

3 Probabilistic Analysis of Kernels Transformed with Loop Perforation

This chapter presents probabilistic accuracy analysis for four computational patterns that interact well with loop perforation. We described the structural patterns that appeared in benchmark applications in Section 2.4. The *sum* pattern calculates the sum of elements. The *mean* pattern calculates the mean of elements. The *argmin-sum* pattern calculates the index of the minimum sum of elements. The *ratio* pattern, calculates the ratio of two sums.

The patterns describe the structure of the computation, the locations in the code that are amenable to transformation, and the pattern's inputs and outputs. For example, the following code represents a mean pattern:

```
val sum = 0.0;
for (int i = 1; i <= n; i++) {
     sum += f(i);
}
val mean = sum / n;
return mean;
```

The pattern abstracts away the details of the computation performed in each loop iteration by representing the value computed by the loop body as a function call `f(i)`. We call each such abstracted value an *input* to the pattern. The *output* of the computational pattern is the variable `mean`.

Loop perforation can change the shaded parts of the code to skip a fraction of iterations `r`. In addition to skipping iterations, we extend the definition of loop perforation to perform an optional *bias reduction* transformation. Bias reduction modifies the output of a pattern to remove systematic bias of the computation by multiplying it with a constant factor. For example, the output `mean` is transformed from `sum / n` to `sum / (r * n)`.

The analysis operates on structural patterns that are amenable to loop perforation. We now describe the main components of the analysis.

Local error analysis. The analysis quantifies the effect of loop perforation as the difference between the results produced by the exact computation (without perforation) and perforated computation (with perforation rate r). In this chapter, we refer to this difference as *perforation noise* and denote it as $D(r)$. For example, if the execution of mean computation produces the value `mean` in the exact version and the value $\texttt{mean}'_r$ in the perforated version of the computation, then perforation noise is $D_{mean}(r) = \texttt{mean} - \texttt{mean}'_r$. We define perforation noise similarly for the other computational patterns.

Probabilistic model of uncertainty. We use random variables to model our *uncertainty* about the computation's inputs and the results. We express the perforation noise as a function of these random variables. Specifically, the analysis models the result of each loop body (represented by a call to `f(i)`) as a random variable X_i.

The perforation noise $D(r)$, which is a function of the values computed by the loop body, is therefore also a random variable. In general, random variables X_i in our analyses can represent 1) inherent randomness in the inputs and/or 2) the developer's incomplete knowledge about the exact underlying processes that produce these inputs. This probabilistic model therefore allows analyzing both probabilistic computations (when the pattern's inputs `f(i)` are random quantities) and deterministic computations (when the pattern's inputs `f(i)` are not necessarily random quantities, but the developer represents his or her partial understanding about the computation that produces the pattern's inputs as a probability distribution of its output).

Specification

The analysis takes a specification of inputs, specification of the accuracy of the computation's output, and the desired perforation rate.

Input Specification. It specifies full distribution or properties of the random variables $X_1, X_2, \ldots, X_n$, which represent the pattern's inputs. For instance, the variables can be independent and identically distributed (i.i.d.), each with the mean μ and variance σ^2.

Transformation Specification. The analysis takes as inputs the loop perforation strategy (e.g., sampling, truncation, or random) and perforation rate, r (percentage of skipped loop iterations).

Output Specification. The probabilistic output specifications represents bounds of the following quantities:

- **Expected noise:** $\mathbb{E}(D(r)) < B_E$. A developer provides a numerical constant B_E, which is an upper bound on the expected perforation noise. The analysis calculates an overapproximation of $\mathbb{E}(D(r))$.
- **Variance of noise:** $\mathrm{Var}(D(r)) < B_V$. A developer provides a numerical constant B_V, which is an upper bound on the variance of perforation noise. The analysis calculates an overapproximation of $\mathrm{Var}(D(r))$
- **Probability of large noise:** $\Pr(|D(r)| \geq B_P) \leq \varepsilon$. A developer provides the pair (B_P, ε), which specifies 1) B_P, an upper bound on the acceptable absolute perforation noise and 2) ε, an upper bound on the probability of observing noise greater than B_P. The analysis calculates an overapproximation of the probability $\Pr(|D(r)| \geq B_P)$.

Analysis

For each pattern, the analysis produces algebraic expressions that characterize the expected value and variance of the perforation noise, and the probability of observing large absolute perforation noise. We use properties of random variables, in combination with the applied transformations, to derive algebraic expressions that characterize the perforation noise.

We manually derive perforation noise for a number of representative combinations of input and transformation specifications. Conceptually, for each pattern, the analysis maintains a *dictionary* of algebraic expressions for calculating noise bounds. Each of these expressions is parameterized by the perforation rate and the properties of the input random variables (e.g., their mean or variance). In the remainder of this chapter, we present derivations of representative perforation noise expressions for various input assumptions.

The analysis operates in three steps. In the first step, the analysis identifies the structure of the computation. To identify the structural patterns, it is possible to use existing program analyses, such as classical reduction recognition [49, 57] or recent pattern identification procedure from [102]. As a result, this auxiliary analysis returns the pattern, with marked pattern's input and the output variable.

In the second step, the analysis matches the input and transformation specifications with the set of assumptions for each derived algebraic bound that the analysis' dictionary

```
double dPrice = 0.0;
for (int i = 1; i <= lTrials; i += 1) {
   double simres = runSimulation(this, i)
   dPrice += simres;
}
double dMeanPrice = dPrice / lTrials;
printf("%g\n", dMeanPrice);
```

(a) Original Computation

```
double dPrice = 0.0;
for (int i = 1; i <= lTrials; i += 2) {
   double simres = runSimulation(this, i)
   dPrice += simres;
}
double dMeanPrice = (dPrice * 2) / lTrials;
printf("\%g\n", dMeanPrice);
```

(b) Transformed Computation

Figure 3.1: Original and Transformed Swaptions Code

contains. This step returns the matched expression (if it exists) or indicates that the analysis cannot continue with the current specifications.

In the third step, the analysis calculates the noise bound by substituting the input and transformation parameters from the developer's specification. Finally, the analysis checks if this computed bound is smaller than the developer-provided acceptable noise bound from the output specification.

Sources. The previous version of the research presented in this chapter appeared in [78] and accompanying technical report [77].

3.1 Motivating Example

Swaptions is a financial analysis application from the PARSEC benchmark suite [11]; it uses Monte Carlo simulation to calculate the price of a portfolio of swaption financial instruments. Figure 3.1(a) presents a simplified version of a perforatable computation from this application. The loop performs a sequence of simulations to produce an estimated swaption value `dMeanPrice`.

Figure 3.1(b) presents the transformed version of the computation after applying loop perforation [52]. The transformed expressions are shaded. The transformation changes the induction variable increment from `i += 1` to `i += 2`. The perforated loop therefore executes half of the iterations of the original loop. The transformed computation also extrapolates the result by doubling `dPrice` to eliminate a systematic bias introduced by executing fewer loop iterations.

Pattern Structure

This computation as an instance of the *mean* pattern. The variable `simres` is the input of the pattern. The variable `dMeanPrice` is the output of the pattern. The number of loop iterations is `n`. The analysis models the values of the program variable `simres` during the execution of the loop as a sequence of random variables $X_1, X_2, \ldots, X_n$. Therefore, the program variables `dPrice` and `dMeanPrice`, which are derived from the value of `simres`, are also modeled as random variables.

The program variable `dPrice` contains the sum of the X_i. In the original computation, the analysis represents the value of `dMeanPrice` is $S_O = \frac{1}{n}\sum_{i=1}^{n} X_i$. In the perforated computation, the value of `dMeanPrice` is $S_P = \frac{2}{n}\sum_{i=1}^{n/2} X_{2i-1}$ (for notational simplicity, we present the analysis for the case when `n` is even). The perforation noise is $D = S_P - S_O$.

Specifications

We have developed analyses to support different input specifications. The specification states different properties of the distributions of the random variables $X_1, \ldots, X_n$ and the number of loop iterations `n`. Below are some instances of input specifications:

- *I.I.D. Inputs :* All $X_1, \ldots, X_n$ are independent and identically distributed (i.i.d.) with the mean μ and variance σ^2.
- *I.I.D. and Bounded Inputs:* All $X_1, \ldots, X_n$ are random variables bounded on the interval $[a, b]$. A special case is when they the variables are uniformly distributed.
- *I.I.D. and Gaussian Inputs:* All $X_1, \ldots, X_n$ are i.i.d. $N(\mu, \sigma^2)$ variables.
- *Correlated Random Walk:* The variables $X_1, \ldots, X_n$ are correlated, and there exist random noise variables $\xi_1, \ldots, \xi_{n-1}$, such that $X_{i+1} = X_i + \xi_i$. All random noise variables $\xi_1, \ldots \xi_{n-1}$ are i.i.d. with mean 0 and variance σ^2

The developer also provides 1) the transformation specification, which specifies the perforation strategy (e.g., interleaving), and perforation rate (e.g., skip half iteration). and 2) the output specification, e.g., the bound on the variance of the pattern's result, or the bound on the probability of large deviations.

As a concrete example for swaptions loop, consider:

- **Input Specification:** All inputs (the prices of the swaptions) are modeled as i.i.d. random variables in the interval $[a, b] = [0.0, 1.0]$. The number of original iterations of the loop is $n = 20000$.

- **Transformation Specification:** The transformation is interleaving perforation with perforation rate $r = 0.5$.
- **Output Specification:** The outputs should have absolute output deviation smaller than $B_P = 0.01$ (one cent) with probability at least 0.95. This is equivalent to saying that the outputs can have perforation noise greater than $B_P = 0.01$ with probability at most $\varepsilon = 0.05$, i.e., $P(|D| \geq B_P) \leq \varepsilon$.

Analysis

For each input and output specification, the analysis computes the perforation noise bound and compares it to the one provided by the developer. We show the derivation of the expressions in Sections 3.3 to 3.6.

Given the developer's specification, the analysis will first match input specification to the dictionary of the analyzable input specifications. The analysis will then compute the expression that represents the upper bound of the quantity of interest (which may be variance, mean, or perforation noise).

For the swaptions example, the analysis computes the maximum bound on the perforation noise, $\bar{B}_P$, that satisfies the input specification (the random inputs within the interval $[a, b]$), transformation specification (perforation noise r, and the number of iterations of perforated loop $m = \lfloor r\, n \rfloor$), and the acceptable probability of large deviation from the output specification, ε.

The analysis calculates the bound $\bar{B}_P$ using the following expression:

$$\bar{B}_P = \sqrt{\frac{1}{2} \ln \frac{2}{\varepsilon} \cdot \sum_{i=1}^{n} \left(b_i^* - a_i^*\right)^2} = (b-a)\sqrt{\frac{n-m}{2\,m\,n} \ln \frac{2}{\varepsilon}}$$

We present the derivation of expressions of this form in Sections 3.3.4 and 3.4.

The analysis uses the inputs from the developer's specification to calculate $\bar{B}_P = 0.0096$. Finally, the analysis compares this value to B_P. Since $\bar{B}_P = 0.0096 \leq 0.01 = B_P$, the perforated computations satisfies the developer's output specification.

Comparison with Worst-Case Bound. Finally, we can compare this bound with the worst-case bound. The maximum error bound is (Section 2.6) $B_W = \frac{n-m}{m}(b-a) = \frac{b-a}{2}$. The worst-case bound is therefore equal to 0.5, compared to the 95% bound of 0.0096. If we calculate the maximum noise bounds for some other probabilities, we get e.g., that 99% bound is 0.012, 99.9% bound is 0.14, they help developer understand the behavior of the

perforated computation. Overall, the worst case bound B_W is asymptotically $\sqrt{n}$ times larger than the probabilistic bound B_P.

3.2 Preliminaries

In this section we present a formal definition of loop perforation that we will use in the analysis and review several well-known probabilistic inequalities.

3.2.1 Pattern Components

In the rest of this chapter we present the analyses for the computational patterns. We describe now the outline of each of the sections:

Pattern Example. For each pattern, we present an example of the original and the transformed code. In the examples, we apply an interleaving perforation (with the `n` iterations) of perforatable loops. We use the increment `k`, which is derived from the perforation rate (see Section 3.2.2).

Pattern Structure. For each pattern, we identify its inputs and outputs, identify the part of computation to perforate, and define random variables that model our uncertainty about their values.

Input Specification and Analysis. In each pattern analysis section we first present the input specifications that represent assumptions we make on the distribution of the inputs. These assumptions characterize our uncertainty about the values of these inputs.

The analysis description presents the derivation of theq expressions for 1) the mean perforation noise, 2) the variance of the perforation noise, and 3) bounds on the probability of observing large absolute perforation noise. We present multiple analyses, for different input specifications. We start each section with the most general set of assumptions and subsequently present the additional specifications and (more precise) analysis results for those specifications.

Output Specification and Checking. For the output specification, we assume that a developer provides the upper bounds on the expected noise, variance of noise, and the probability of observing output noise larger than a specified bound. The checking procedure calculates the expressions from the analysis and ensures that the calculated quantities are smaller than the developer-provided bounds. For example, if the analysis returns the expected noise bound $\bar{B}_E$, the checker ensures that $\bar{B}_E \leq B_E$.

3.2.2 Definition of Loop Perforation

The original loop executes n iterations, The perforated loop executes m, $m < n$ iterations. The *perforation rate* determines the number of iterations of the perforated loop. If the perforation rate is r, then $m = \lfloor r \cdot n \rfloor$.

A loop perforation strategy can be represented by a $n \times 1$ *perforation vector* P, each element of which corresponds to a single loop iteration. The number of non-zero elements of P is equal to m. The vector $\mathbf{P}$ is known before the execution of the loop. We will denote the perforation vector with all-ones as $\mathbf{A} = (1, \ldots, 1)'$. The vector $\mathbf{A}$ represents the original (non-perforated) computation.

We can define different perforation strategies using the notion of perforation vector:

- **Interleaving perforation** executes every k-th iteration, where $k = \lfloor \frac{n}{m} \rfloor$. The corresponding perforation vector has elements $P_{ki+1} = 1$, where $i \in \{0, ...m-1\}$, and $P_{ki+j} = 0$ for $j < k, j \neq 1$.
- **Truncation perforation** executes m iterations at the beginning of the loop; the perforation vector has elements $P_i = 1$ for $1 \leq i \leq m$, and $P_i = 0$ otherwise.
- **Randomized perforation** selects a random subset of m elements; the perforation vector has elements $P_i = 1$ for a random combination of indices $\{i_1, \ldots, i_m\}$ and 0 for the remaining

Some transformations may further specialize the definition of the perforation vector. For instance, the perforation transformation for the sum pattern changes the values of the vector's elements to define bias reduction. Specifically, it multiplies the vector $\mathbf{P}$ with a value that can appropriately extrapolate the result of the sum computation.

3.2.3 Useful Probabilistic Inequalities

In this section we describe a few useful inequalities that can help us calculate the probability of large perforation noise.

Markov's inequality. For a random variable Y with finite mean, Markov's inequality provides an upper bound on the probability of observing its absolute value exceed a. In particular,

$$\Pr\Big(|Y| \geq a\Big) \leq \frac{\mathbb{E}(|Y|)}{a}. \tag{3.1}$$

This inequality is non-trivial when $a > \mathbb{E}(|Y|)$.

Chebyshev's inequality. For a random variable Y with finite mean and variance, Chebyshev's inequality gives an upper bound on the probability of observing that an absolute difference between Y and its mean, $\mathbb{E}(Y)$, is greater than some value a. In particular,

$$\Pr\Big(|Y - \mathbb{E}(Y)| \geq a\Big) \leq \frac{\mathrm{Var}(Y)}{a^2}. \tag{3.2}$$

This inequality is non-trivial when $a^2 > \mathrm{Var}(Y)$.

Hoeffding's inequality. For a sum of random variables $S_n = X_1 + \ldots + X_n$ where all terms X_i are independent and almost surely bounded, i.e., $P(X_i \in [a_i, b_i]) = 1$, Hoeffding's inequality gives an upper bound on the absolute difference between S_n and its mean larger than some constant t. In particular,

$$\Pr\Big(\Big|S_n - \mathbb{E}(S_n)\Big| \geq t\Big) \leq 2 \cdot \exp\Big(-\frac{2 \cdot t^2}{\sum_{i=1}^{n}(b_i - a_i)^2}\Big). \tag{3.3}$$

Because of the additional assumptions on the input random variables, Hoeffding's inequality often provides tighter bounds than Chebyshev's or Markov's inequality.

3.3 Sum Pattern

We present an example of the original and perforated code for the extrapolated sum pattern in Figure 3.2. We first present a generalized analysis for the sum of correlated random variables. We then present specializations of the analysis under additional assumptions. Special cases that we analyze include independent and identically distributed (i.i.d.) inputs and inputs generated by a random walk.

Original code	Transformed Code

`double sum = 0.0;`
`for (int i = 1; i <= n; i++) {`
`  sum += f(i);`
`}`
	`double sum = 0.0;`
	`for (int i = 1; i <= n; i+=k) {`
	`  sum += f(i);`
	`}`
	`sum *= k;`

Figure 3.2: Sum Pattern; Original and Transformed Code

Pattern Structure. The inputs of the pattern are the results of the function `f(i)`. The analysis represents these inputs as random variables X_i. The output of the pattern is the variable `sum`. Since it is computed using the inputs $X_1, \ldots, X_n$, it is also a random variable. Perforation noise D is the difference between the values of the variable `sum` in the original and perforated versions of the code.

The transformed pattern performs a systematic bias reduction, by multiplying the result of the computation with the constant proportional to the number of skipped iterations.

3.3.1 General Inputs

Input Specifications. We first assume only that the terms of the sum have a common finite mean μ and finite covariance, expressed with covariance matrix Σ.

Analysis. For $i = 1, \ldots, n$, let $X_i = \mathtt{f}(i)$ be the i-th term of the summation. The analysis represents the uncertainty about the values X_i by defining the vector $\mathbf{X} = (X_1, \ldots, X_n)'$ as a vector of n random variables. Each random variable has the mean μ and covariance matrix Σ with elements $(\Sigma)_{ij} = \mathrm{cov}(X_i, X_j)$, which are obtained from the input specification.

Let $\mathbf{A}$ be the all-ones perforation vector defined in Section 3.2.2. Then $\mathbf{A}'\mathbf{X} = \sum_{i=1}^{n} X_i$. Let $\mathbf{P}$ be a perforation vector with m non-zero elements. Then $\mathbf{P}'\mathbf{X} = \sum_{i=1}^{n} P_i \cdot X_i$ is the result of the perforated computation.

The signed perforation noise is $D \equiv \mathbf{P}'\mathbf{X} - \mathbf{A}'\mathbf{X} = (\mathbf{P} - \mathbf{A})'\,\mathbf{X}$. We can therefore immediately compute the expected value and variance of noise,

$$\mathbb{E}(D) = \mu \sum_{i=1}^{n} (P_i - 1), \tag{3.4}$$

$$\mathrm{Var}(D) = \sum_{i,j} (P_i - 1)\,(P_j - 1)\,\Sigma_{i,j}. \tag{3.5}$$

To avoid systematic bias, we need to choose $\mathbf{P}$ so that $\mathbb{E}(D) = 0$. In particular, it follows from Equation 3.4 that an estimate is *unbiased* if and only if $\sum_{i=1}^{n} P_i = n$. One extrapolation strategy equally extrapolates every non-zero element, choosing $P_i = \frac{n}{m}$ for non-zero elements P_i.

If $\mathbf{P}$ satisfies $\mathbb{E}(D) = 0$, we can use Chebyshev's inequality and $\mathrm{Var}(D)$ to bound the absolute perforation noise (for some bound B_P and probability ε from the developer's

output specification):

$$\Pr\Big(|D - \mathbb{E}(D)| \geq B_P\Big) \;\leq\; \frac{\mathrm{Var}(Y)}{B_P^2} \;\leq\; \varepsilon.$$

Therefore, with probability at least $1 - \varepsilon$,

$$|D| < B_P = \sqrt{\frac{\mathrm{Var}(D)}{\varepsilon}} \tag{3.6}$$

This bound will be conservative in practice. Additional knowledge (e.g., independence or a specific distribution of the variables X_i) can be used to derive tighter bounds. We next study a number of special cases in which additional assumptions enable us to better characterize the effect of perforation.

3.3.2 Independent Inputs

Input Specifications. We assume that the inputs $X_i = \mathtt{f}(i)$ of the summation are i.i.d. random variables with finite mean μ and variance σ^2.

Analysis. From (3.4), we know that $\mathbb{E}(D) = 0$ for any perforation P such that $\sum_i P = n$. From (3.5), and since the covariance matrix Σ of i.i.d. variables has non-zero values only along its leading diagonal, it follows that $\mathrm{Var}(D) = \sigma^2 \sum_i (1 - P_i)^2$. It is straightforward to show that this value is minimized by any perforation vector P with $n - m$ zeros and the remaining elements taking the value $\frac{n}{m}$. In this case, the variance takes the value

$$\mathrm{Var}(D) = \frac{\sigma^2\, n\,(n - m)}{m}. \tag{3.7}$$

We can immediately bound the probability of observing large absolute perforation noise using Chebyshev's inequality (Equation 3.6).

3.3.3 Independent Gaussian Inputs

Input Specification. We assume that the variables X_i are i.i.d. with Gaussian distribution, with finite mean μ and variance σ^2.

Analysis. In this case we can get potentially tighter bounds than in the previous analysis for independent variables. Since each X_i is normally distributed, D will also be normally distributed. Consequently, $\mathbb{E}(D) = 0$ and $\mathrm{Var}(D)$ remains the same as in Equation 3.7.

The normality of D allows us to obtain a tighter bound on the perforation noise. In particular, with probability $1 - \varepsilon$

$$|D| \leq z_{1-\frac{\varepsilon}{2}} \sqrt{\mathrm{Var}(D)} \tag{3.8}$$

where z_α is the quantile function of the standard normal distribution. As a comparison, for $\varepsilon = 0.01$ the bound (3.8) is 6.6 times smaller than the bound obtained from Chebyshev's inequality (3.6).

3.3.4 Independent Bounded Inputs

Input Specification. We assume that the variables X_i are i.i.d. and all are bounded within the interval $[a, b]$, i.e., $\forall i \,.\, \Pr(X_i \in [a, b] = 1)$.

Analysis. In this case, we can compute the bound on large absolute perforation noise $|D|$. We present how we derive this bound using Hoeffding's inequality (3.3).

Since the variables are bounded, their mean is also bounded. Therefore, to get the unbiased result ($\mathbb{E}(D) = 0$), we can use the same argument as for the general analysis in Section 3.3.1. Therefore, the perforation vector $\mathbf{P}$ has m zero elements and all other non-zero elements have the values $P_i = \frac{n}{m}$.

We define $X_i^* = (P_i - 1)X_i$, and note that the variables X_i^* are also mutually independent and bounded. The range of X_i^* is $[a_i^*, b_i^*] = \big[(P_i - 1)a, (P_i - 1)b\big]$. Then, $\sum_{i=1}^{n}(b_i^* - a_i^*)^2 = (b-a)^2 \cdot \frac{n\,(n-m)}{m}$.

We can replace the previous sum in the Hoeffding's inequality, and therefore with probability at least $1 - \varepsilon$,

$$|D| < \sqrt{\frac{1}{2}\ln\frac{2}{\varepsilon} \cdot \sum_{i=1}^{n}\left(b_i^* - a_i^*\right)^2} = (b-a)\sqrt{\frac{n\,(n-m)}{2m}\ln\frac{2}{\varepsilon}}. \tag{3.9}$$

3.3.5 Random Walk

Input Specifications. We assume that the sequence X of random variables is a random walk with independent increments. Specifically, we assume that the sequence is a Markov process, and that the differences between the values at adjacent time steps $\xi_i = X_{i+1} - X_i$ are a sequence of i.i.d. random variables with mean 0 and variance σ^2. Let $X_0 = \mu$ be a constant.

Analysis. From the assumption $\mathbb{E}(\xi_i) = 0$, it follows by induction that the expected value of every element is $\mathbb{E}(X_i) = \mu$. As a consequence, for any perforation vector that satisfies $\sum_{i=1}^{n} P_i = n$, we have that $\mathbb{E}(D) = 0$.

For $i < j$, the covariance between X_i and X_j satisfies $cov(X_i, X_j) = i\sigma^2$. Therefore, the covariance matrix Σ has entries $(\Sigma)_{ij} = \sigma^2 \min\{i, j\}$, and the variance of the perforation noise satisfies

$$\mathrm{Var}(D) = \sigma^2 \sum_{i,j} (1 - P_i)(1 - P_j) \min\{i, j\}. \tag{3.10}$$

3.4 Mean Pattern

We present an example of the original and perforated code in Figure 3.3.

Original code	Transformed Code

`double sum = 0.0;`
`for (int i = 1; i <= n; i++) {`
`  sum += f(i);`
`}`
`double mean = sum / n;`
	`double sum = 0.0;`
	`for (int i = 1; i <= n; i+=k) {`
	`  sum += f(i);`
	`}`
	`double mean = sum * k / n;`

Figure 3.3: Mean Pattern; Original and Transformed Code

This pattern extends the sum pattern. Specifically, the difference is that the output of the mean pattern, the variable `mean` divides the variable `sum` from the sum pattern by the number of loop iterations `n`.

We can extend the analysis for the sum pattern (Section 3.3) because the result of the mean computation is equal to the result of the sum computation divided by n. We denote the perforation noise of the sum as D_{Sum}, the output produced by the original computation as $\frac{1}{n}A'X$, and the output produced by the perforated computation as $\frac{1}{n}P'X$. The perforation noise of the mean D in the general case with correlated variables is $D \equiv \frac{1}{n}(P'X - A'X) = \frac{1}{n}D_{Sum}$. By the linearity of expectation, the perforation noise has expectation $\mathbb{E}(D) = \frac{1}{n}\mathbb{E}(D_{Sum})$ and variance

$$\mathrm{Var}(D) = \frac{1}{n^2}\mathrm{Var}(D_{Sum}). \tag{3.11}$$

The derivation of the bounds for the more specific cases (i.i.d., normal, random walk inputs) is analogous to the derivation discussed in Section 3.3.

3.5 Argmin-Sum Pattern

We present an example of the original and transformed code for the argmin-sum pattern in Figure 3.4. We can apply the same analysis to the *min-sum* pattern, which returns the value `best` instead of the index `best_index`. It is also possible to slightly modify this analysis to support the *max-sum* and *argmax-sum* patterns (by using the identity $\max(a, b) = -\min(-a, -b)$).

Original code	Transformed Code

`double best = MAX_DOUBLE;`
`int best_index = -1;`
`for (int i = 1; i <= L; i++) {`
`  s[i] = 0;`
`  for (int j = 1; j <= n; j++)`
`  s[i] += f(i,j);`
 	
`  if (s[i] < best) {`
`  best = s[i];`
`  best_index = i;`
`  }`
`}`
`return best_index;`
	`double best = MAX_DOUBLE;`
	`int best_index = -1;`
	`for (int i = 1; i <= L; i++) {`
	`  s[i] = 0;`
	`  for (int j = 1; j <= n; j+=k)`
	`  s[i] += f(i,j);`
	
	`  if (s[i] < best) {`
	`  best = s[i];`
	`  best_index = i;`
	`  }`
	`}`
	`return best_index;`

Figure 3.4: Argmin-Sum Pattern; Original and Transformed Code

Pattern Structure. The inputs of the pattern are the values produced by the function `f(i,j)`. The analysis represents each call to the function with the random variable $X_{i,j}$. The outer loop (which cannot be perforated) executes for L iterations, the inner loop (which can be perforated) executes for n iterations. The computation produces two outputs, the minimum sum `best` and the index of the array element that corresponds to this sum `best_index`.

The analysis characterizes the accuracy of the computation via the sum `best`, which is used to rank the different elements. Since `best` is computed from the inputs that are random, it is also a random variable, that we denote as M. The rationale for choosing `best` for quantifying the accuracy of the pattern is that the remaining computation is expected produce a similar result if the element returned by the perforated computation is similar to the element returned by the original computation, even though the indices can have arbitrary distance.

Input Specifications. For each $i \in \{1, \ldots, L\}$, we assume that $X_{i,j} = \texttt{f(i,j)}$ are independent and identically distributed, with an finite mean μ.

Analysis. The best score of element returned by the original computation is $M_O = \min_{i=1:L} \sum_{j=1}^{n} X_{i,j}$. But, to facilitate the analysis, we will represent this expression equivalently as

$$M_O = \sum_{j=1}^{n} X_{\omega,j} \qquad \text{with the index} \qquad \omega \equiv \arg\min_{i=1:L} \sum_{j=1}^{n} X_{i,j}.$$

Analogously, the best score of the element returned by the perforated computation is

$$M_P = \sum_{j=1}^{n} X_{\gamma,j} \qquad \text{with the index} \qquad \gamma \equiv \arg\min_{i=1:L} \sum_{j=1}^{m} X_{i,j}.$$

Note that the index γ is computed by iterating only over m elements in the perforated sum[1]. However, M_P is a sum of n variables – this is the *true* score for the element, although the perforated code does not compute this value explicitly.

The analysis computes the perforation noise $D \equiv M_P - M_O$. The perforation noise D is non-negative because M_O is a minimum sum. To represent the noise as a function of perforation, we define additional derived variables. Let $Y_i \equiv \sum_{j=1}^{m} X_{i,j}$ be the partial score computed by the perforated inner loop and $Z_i \equiv \sum_{j=m+1}^{n} X_{i,j}$ be the remainder of the score. Then, $M_O = Y_\omega + Z_\omega$ and $M_P = Y_\gamma + Z_\gamma$.

Then the perforation noise satisfies

$$D = Y_\gamma + Z_\gamma - Y_\omega - Z_\omega \tag{3.12}$$

$$\leq Y_\gamma + Z_\gamma - \min_{i=1:L} Y_i - \min_{i=1:L} Z_i \tag{3.13}$$

$$\leq Z_\gamma - \min_{i=1:L} Z_i. \tag{3.14}$$

The inequality follows from the fact that $M_O = \min_{i=1:L} (Y_i + Z_i) \geq \min_{i=1:L} Y_i + \min_{i=1:L} Z_i$. The fact that $Y_\gamma = \min_{i=1:L} Y_i$ follows from the definitions of γ and Y_i.

Let $\bar{D} \equiv Z_\gamma - \min_i Z_i$ denote this upper bound. We can obtain conservative estimates of the perforation noise D by studying $\bar{D}$. Note that for this pattern, $\bar{D}$ is always non-negative (because $Z_\gamma \geq \min_i Z_i$).

Since the inputs $X_{1,1}, \ldots, X_{L,n}$ are mutually independent, then so are the partial sums $Z_1, \ldots, Z_L$. Since, by definition, Z_γ is one of these sums, then it is independent from

[1]The independence of the variables $X_{i,j}$ implies that any perforation vector $\mathbf{P}$ that selects m elements will result in the same expected error and variance. We can therefore, without loss of generality, continue the analysis with the strategy that executes the first m iterations (as we defined above).

the remaining partial sums. Since each of them sums $n - m$ elements, the expectation $\mathbb{E}(Z_i) = (n - m) \cdot \mu$. The expectation of $\bar{D}$ is therefore (from the linearity of expectation):

$$\mathbb{E}(\bar{D}) = (n - m) \cdot \mu - \mathbb{E}(\min_{i=1:L} Z_i).$$

This expression provides a general upper bound on the noise D. Since D is positive, we can in principle use Chebyshev's bound to limit the probability of large absolute perforation noise. However, to provide an analytic expression for the expected value $\mathbb{E}(\min_{i=1:L} Z_i)$, we need to make additional assumptions on the distribution of the inputs $X_{i,j}$ or the distribution of the remainder sums Z_i.

3.5.1 Gaussian Inputs

Input Specification. We assume that the inputs $X_{i,j}$ are i.i.d Gaussian variables, with the mean μ and variance σ^2.

Analysis. Since $X_{i,j}$ are i.i.d Gaussians, then so are their partial sums Z_i (but with mean $(n - m)\mu$ and variance $(n - m)\sigma^2$. Therefore, $\mathbb{E}(\min_i Z_i)$ for $i \in \{1, \ldots, L\}$ is the minimum of the Gaussian variables.

We compute a lower bound of this expectation, by computing an upper bound of the expectation $\mathbb{E}(\max_i Z'_i)$, where each $Z'_i = -Z_i$ is a Gaussian variable with the mean $-(n - m)\mu$. We derive this bound as follows. Let $V = \max_i Z'_i$. Then $\exp(\mathbb{E}(V)) \leq \mathbb{E}(\exp(V))$ (from Jensen's inequality). Also, $\exp(V) = \max_i \exp(Z'_i)$ and this is bounded from above by $\sum_{i=1}^{L} \exp(Z'_i)$ (since the expressions under max are non-negative). Since we have that $Z'_i \sim N(-(n - m)\,\mu, (n - m)\,\sigma^2)$, a random variable $\exp(Z'_i)$ has a log-normal distribution, and therefore its expected value is $\exp(-(n - m)\mu + (n - m)\sigma^2/2)$. There are L such independent log-normal variables, which entails $\exp(\mathbb{E}(V)) \leq L \exp(-(n - m)\mu + (n - m)\sigma^2/2)$. Finally, we can derive the bound $\mathbb{E}(V) \leq \log(L) - (n - m)\mu + (n - m) \cdot \sigma^2/2$. Therefore, we have bound:

$$\mathbb{E}(D) \leq \log(L) + (n - m) \cdot \sigma^2/2.$$

Since D is always positive (i.e., $D = |D|$), we can use Markov's inequality to bound the probability of large absolute perforation noise. However, for the Gaussian variables, variance of the perforation noise does not have a closed form [7].

3.5.2 Analysis for Approximate Assumptions

Another way to proceed with the analysis is to directly make an assumption on the distribution of the variables $Z_1, \ldots, Z_L$. In such cases, the analysis is likely to be *approximate*

in the sense that it will not provide guarantees for the input specification. However, these analyses can be potentially applied if sensitivity profiling identifies that the empirical distribution of data is similar to one of these distributions. For example, we anticipate that Z_i will in practice rarely be uniform, however this assumption is in some sense conservative if we choose the center and width to cover all but a tiny fraction of the mass of the true distribution of Z_i.

Uniform distribution of Z's. We make the assumption that all Z_i are i.i.d. and uniformly distributed on the interval $a \pm \frac{w}{2}$ of width $w > 0$ and center a.

If $W_L = \min_{i \le L} Z_i$, then $\frac{1}{w}(W_L - a + \frac{w}{2})$ has a Beta$(1, L)$ distribution, and so $\mathbb{E}(W_L) = a + \frac{w}{L+1} - \frac{w}{2}$ and variance $\mathrm{Var}(W_L) = \frac{Lw^2}{(L+1)^2(L+2)}$. From (3.15), we have $\mathbb{E}(\bar{D}) = \frac{w}{2} - \frac{w}{L+1}$. Furthermore, as γ is independent of every Z_i, it follows that Z_γ is independent of W_L. Therefore,

$$\mathrm{Var}(D) \le \mathrm{Var}(\bar{D}) = \frac{1}{12}w^2 + \frac{Lw^2}{(L+1)^2(L+2)}.$$

From this expression, we can bound the probability of large perforation noise using Chebyshev's inequality.

Exponential distribution of Z's. We make the assumption that all Z_i are i.i.d. and exponentially distributed with the rate parameter λ. Then, the minimum of these variables is also an exponential random variable, with the expected values $\mathbb{E}(\min_{i=1:L}(Z_i)) = (L \cdot \lambda)^{-1}$. From this and the fact that $\mathbb{E}(Z_i) = \lambda^{-1}$, the bound on the expected perforation noise (from Equation 3.15) is

$$\mathbb{E}(D) \le \frac{L-1}{L \cdot \lambda}.$$

3.6 Ratio Pattern

We present an example of the original and transformed code in Figure 3.5.

Input Specifications. Let $X_i = \mathbf{x}(i)$ and $Y_i = \mathbf{y}(i)$ denote random variables representing the values of the inner computations. We assume that the sequence of pairs (X_i, Y_i) are i.i.d. copies of a pair of random variables (X, Y), where $Y > 0$ almost surely (i.e., $\Pr(Y > 0) = 1$). Define $Z = X/Y$ and $Z_i = X_i/Y_i$. For some constants μ and σ_Z^2, we assume that the conditional expectation of Z given Y is μ, i.e., $\mathbb{E}(Z|Y) = \mu$, and that the conditional variance satisfies $\mathrm{Var}(Z|Y) = \frac{\sigma_Z^2}{Y}$.

Original code	Transformed Code

`double numer = 0.0;`
`double denom = 0.0;`
`for (int i = 1; i <= n; i++) {`
`  numer += x(i);`
`  denom += y(i);`
`}`
`return numer/denom;`
	`double numer = 0.0;`
	`double denom = 0.0;`
	`for (int i = 1; i <= n; i+=k) {`
	`  numer += x(i);`
	`  denom += y(i);`
	`}`
	`return numer/denom;`

Figure 3.5: Ratio Pattern; Original and Transformed Code

Analysis. The elements of the perforation vector $\mathbf{P}$ only take values from the set $\{0, 1\}$. The independence of the pairs (X_i, Y_i) from different iterations implies that the perforation strategy does not influence the final result. To simplify the derivation, but without loss of generality, we use the perforation vector $\mathbf{P}$ in which the first m elements are 1 and the remaining elements 0.

Define $Y_1^n = A'Y = \sum_{i=1}^n Y_i$ and $Y_1^m = P'Y = \sum_{i=1}^m Y_i$ and define X_1^n and X_1^m analogously. Then the value of the original computation is $S_O = \frac{X_1^n}{Y_1^n} = \sum_{i=1}^n \frac{Y_i}{Y_1^n} Z_i$, while the value of the perforated computation is given by $S_P = \sum_{i=1}^m \frac{Y_i}{Y_1^m} Z_i$, where m is the reduced number of steps in the perforated sum. Note that in the previous equations, we used the identity $X_i = Y_i Z_i$.

We begin by studying the (signed) perforation noise $D \equiv S_P - S_O$. The conditional expectation of D given $Y_{1:n} = \{Y_1, \ldots, Y_n\}$ satisfies $\mathbb{E}(D|Y_{1:n}) = \sum_{i=1}^n \frac{Y_i}{Y_1^n}\mu - \sum_{i=1}^m \frac{Y_i}{Y_1^m}\mu = 0$. The conditional variance satisfies $\mathrm{Var}(D|Y_{1:n}) = \sigma_Z^2 \left(\frac{1}{Y_1^m} - \frac{1}{Y_1^n}\right)$ By the law of iterated expectations $\mathbb{E}(D) = \mathbb{E}(\mathbb{E}(D|Y_{1:n})) = 0$.

3.6.1 Gamma Inputs

Input Specification To proceed with an analysis of the variance of the perforation noise D, we make a distributional assumption on Y. In particular, we assume that Y is gamma distributed with shape $\alpha > 1$ and scale $\theta > 0$.

Analysis. Therefore, the sum Y_1^m also has a gamma distribution with parameters $\alpha' = m\alpha$, $\theta' = \theta$, $\frac{1}{Y_1^m}$ has an inverse gamma distribution with mean $(\theta(m\alpha - 1))^{-1}$, and so

$$\mathrm{Var}(D) = \frac{\sigma_Z^2}{\theta}\left(\frac{1}{m\alpha - 1} - \frac{1}{n\alpha - 1}\right). \tag{3.15}$$

Again, using Chebyshev's inequality, we can bound the probability of large $|D|$.

3.7 Discussion

Scope. This chapter presents probabilistic guarantees for the accuracy of perforated computations. We expect that the basic framework of the probabilistic guarantees (algebraic expressions for expected values, variances, and probabilistic bounds) will remain largely the same for other transformations (the derivation of the expressions will of course differ). For instance, the next chapter presents another set of probabilistic specifications for approximate operations/variables and an automated analysis to check these specifications.

We note that even for loop perforation, we do not attempt to provide an exhaustive list of the possible patterns and analyses. To provide a more complete set of statically analyzable patterns, we note that the statistical literature provides a comprehensive treatment of operations on random variables [115] and order statistics of random variables [7]. The basic compositional properties of probability distributions under such operations can provide the foundation for the analysis of computations which employ many of these operations. In addition, sampling-based analysis techniques, such as [13, 107, 108] can help propagate the transformation error through the subsequent computation (while verifying probabilistic assertions with high confidence). However, a main limitation of all analyses that require a developer to provide full distributional assumptions is that, in general, small changes in the detailed assumptions may significantly change the analysis results.

Representation of Patterns. We represented the structural perforatable patterns as snippets of imperative code written in a C-like language. We note that these patterns can also be represented using different language constructs with the same semantics.

For instance, the sum pattern can be represented using the recursion. If the function that computes each element is `f()`, then the pattern can be represented as

```
sum f 0 = f 0
sum f n = f n + sum f (n-1)
```

The pattern can also be represented using the functional language operators `map` and `fold`,

```
inputs = range(1, n)
sum = fold (0 (+) (map f inputs))
```

The other patterns can be similarly represented using different language constructs. Representation of computation affects the transformation that will be applied, for each of these code patterns we can define a transformation that has the same effect as loop perforation.

Composition of Transformations. In addition to randomness of inputs, we can also successfully study the noise that comes from the computations. An example of such trans-

formation is random loop perforation, which corresponds to random sampling. We also explored other randomized transformations, such as randomized function substitution, which selects one of multiple alternative function implementations. If the transformation is random then, in principle, the analysis can describe the set of inputs using either a distribution or some other deterministic representation (e.g.,intervals).

In our subsequent work, we presented a more general computational model and approximate computation pattern [125]. This model represents a computation as an abstract tree of multiple map tasks (which apply a function on a list of input values) and reduction tasks (which average or compute the minimum/maximum of a list of numbers). The model supports two accuracy-aware transformations: randomized function substitution for map tasks and random sampling for reduction tasks. In addition, the developer provides the input intervals and specifications of accuracy and performance for the alternative function implementations. The analysis then computes the error that emerges and propagates through the model as a function of the probabilities of executing alternative function implementations and probabilities of sampling elements from reduction operators.

In addition, this work formulates the optimization problem that searches for configurations of randomized transformations that deliver profitable accuracy/performance tradeoffs for the computations represented in this abstract computational model [125]. We can also express the optimization problem for the computations we studied in this chapter.

From Analysis to Optimization. Finally, note that the analysis computed probabilistic expressions of probabilistic noise based on a fixed perforation rate. We can instead formulate a problem of *selecting* a perforation rate that satisfies the noise bound, while maximizing performance. To set up the optimization problem, we need an objective function (that characterizes the performance savings) and a constraint (that characterizes perforation noise). Overall, for these loop patterns, the execution time monotonically decreases as we increase perforation rate r. A linear energy estimate is reasonable (especially if different loop iterations execute in approximately the same time). Therefore, we define the objective as $\max r$. We can obtain the constraint from the expressions that we presented in Sections 3.3-3.6, as functions of the unknown perforation noise r. The unknown perforation rate can either be a continuous value $r \in [0, 1]$, or a discrete value (e.g., $r \in \{0.25, 0.5, 0.75\}$).

This optimization problem can be non-convex (because of potentially non-convex constraints. However, for individual loops it is always a univariate optimization problem, and one can use an number of algorithms from numerical analysis (or even derive symbolic expressions) to find the optimum perforation rate. The next three chapters extensively describe how one can define such optimization problems.

4 Reliability-Aware and Accuracy-Aware Optimization with Chisel

Emerging approximate hardware platforms provide operations that may produce less accurate or incorrect results to reduce energy consumption (e.g., [42, 65, 66, 67, 85, 89, 105]). The approximate applications that implement algorithms that are inherently tolerant to inaccuracies in their data and where the majority of the computation is performed in several approximate kernels are good candidates for executing on such hardware devices.

This and the subsequent two chapters presents Chisel, an optimization framework that automatically selects approximate instructions and data that may be stored in approximate memory, given the exact kernel computation and the associated reliability and/or accuracy specification. Chisel can therefore reduce the effort required to develop efficient approximate computations and enhance their portability.

Figure 4.1 presents an overview of Chisel. The developer provides the Chisel program along with reliability and/or accuracy specifications for the approximate kernels. The hardware designer provides a hardware specification, which specifies the reliability and accuracy information for individual instructions and approximate memory.

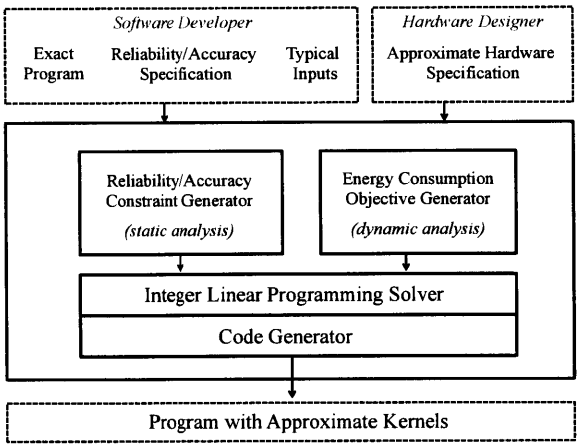


Figure 4.1: Chisel Overview

Exact Program. A Chisel program consists of a kernel function written in the Rely base language [19] (which is a simple imperative language with control-flow constructs and arrays) and code written in an implementation language (such as C) that calls the kernel. The kernel function can compute the return value, but may also write computed values into array parameters passed by reference into the kernel from the outer C code. Chisel transforms the kernel function according to the developer's specification.

Kernel's Reliability and Accuracy Specifications. Reliability specifications of the form `<r*R(x1, ..., xn)>` are integrated into the type signature of the kernel. Here `r` specifies the probability that the kernel (in spite of unreliable hardware operations) computes the value correctly. The term `R(x1, ..., xn)` is a *joint reliability factor* that specifies the probability that `x1,...,xn` all have correct values at the start of the kernel. In the following specification, for example:

```
int <.99 * R(x)> f(int[] <.98*R(x)> x);
```

the return value has reliability at least `.99` times the reliability of the input `x`; when `f` returns, the probability that all elements in the array `x` (passed by reference into `f`) have the correct value is at least `.98` times the reliability of `x` at the start of `f`.

Chisel also supports combined reliability and accuracy specifications of the following form (these specifications are *relational* in that they specify the combined accuracy and reliability with respect to the fully accurate exact computation):

```
<d, r*R(d1 >= D(x1), ..., dn >= D(xn))>
```

Here `d` is a maximum acceptable absolute difference between the approximate and exact result values, `r` is the probability that the kernel computes a value within `d` of the exact value, and the term `R(d1 >= D(x1), ..., dn >= D(xn))` is a joint reliability factor that specifies the probability that each `xi` is within distance `di` of the exact value at the start of the computation. If `r=1`, then the specification is a pure accuracy specification; if `d=0` and all the `di=0`, then the specification is a pure reliability specification.

To support accuracy analysis, the developer can specify the intervals of the inputs using annotations of the form `@interval(x, a, b)`, where `a` and `b` are the lower and the upper bounds of the range of the variable `x`.

Typical Program Inputs. A developer provides a set of typical inputs that Chisel uses to estimate the energy savings of approximate computations. In addition, they can

help developers derive the accuracy and reliability specification through sensitivity profiling (which we describe later in this section).

Approximate Hardware Specifications. Figure 4.2 presents the model of approximate hardware, which consists of approximate ALU, main memory, and cache memory. Chisel works with a hardware specification provided by the designers of the approximate hardware platform [67, 105]. To automatically optimize the implementation of the computation, the optimization algorithm requires a specification of approximate components.

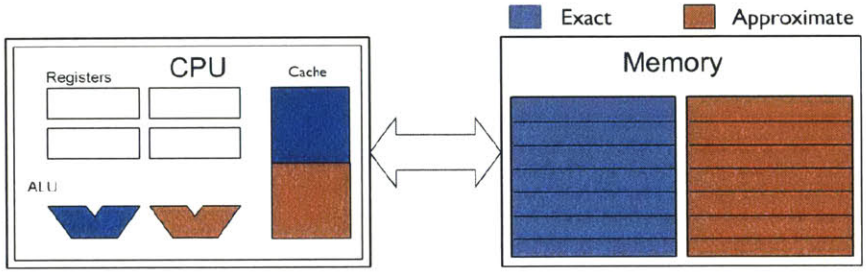


Figure 4.2: Model of Approximate Hardware

The approximate hardware specification consists of:

- **Operation and Memory Accuracy/Reliability.** The hardware specification identifies 1) approximate arithmetic operations and 2) the approximate regions of the main and cache memories. The specification contains the reliability and (optionally) the accuracy loss of each arithmetic operation. It also contains the probability that read and write operations to approximate main memory and cache complete successfully.

- **Energy Model Parameters.** To compute the savings associated with selecting approximate arithmetic operation, the energy model specifies the expected energy savings of executing an approximate version (as a percentage of the energy of the exact version). To compute the savings associated with allocating data in approximate memory, the energy model specifies the expected energy savings for memory cells.

  To compute system energy savings, the energy model also provides 1) a specification of the relative portion of the system energy consumed by the CPU versus memory, 2) the relative portion of the CPU energy consumed by the ALU, cache, and other on-chip resources, and 3) the ratio of the average energy consumption of floating-point instructions and other non-arithmetic instructions relative to integer instructions.

We present examples of approximate hardware specifications later in Chapter 6 (Table 6.1).

Reliability-Aware and Accuracy-Aware Optimization

Chisel's optimization algorithm selects approximate instructions and variables allocated in approximate memories. Chisel automatically navigates the tradeoff space and generates an approximate computation that maximizes energy savings (according to an energy model for the hardware platform) while satisfying its combined reliability and accuracy specification and therefore bound the frequency and magnitude of errors introduced by approximation.

Chisel reduces this search to a numerical optimization problem, which has the following general form:

Minimize:	$\textsc{EnergyConsumption}(\mathit{Configuration})$
Constraints:	$\textsc{Reliability}(\mathit{Configuration}) \geq \mathit{ReliabilityBound}$
	$\textsc{AbsoluteError}(\mathit{Configuration}) \leq \mathit{AccuracyBound}$
Variables:	$\mathit{Configuration} \in \{0,1\}^n$

For each instruction in the kernel, Chisel specifies a *configuration*, a zero-one valued vector that indicates whether each instruction should be exact (zero) or approximate (one). Chisel's optimization algorithm performs static analysis of reliability and accuracy to construct the optimization constraints and dynamic analysis to estimate the energy savings of approximate kernels.

We present Chisel's optimization algorithm in Chapter 5. We show that the optimization problem that the analysis generates is an instance of integer linear programming (ILP), which is a standard problem in mathematical optimization. Chisel can then dispatch the generated problem to an off-the-shelf integer optimization solver and uses the solver's result to generate approximate code that satisfies its reliability and accuracy specifications.

Sensitivity Profiling

To help the developer obtain appropriate reliability and accuracy specifications, Chisel provides an optional sensitivity profiler. The profiler works with an end-to-end sensitivity metric, which compares the end-to-end results of the exact and approximate executions of the Chisel program to define the acceptability requirements for the outputs that the program produces. Specifically, the difference between the exact and approximate executions must be below a specified sensitivity bound. A sensitivity profiler (which performs function-level

noise injection to estimate the sensitivity of the program's result to noise in the results that the kernel computes) can help the developer identify specifications that produce acceptable end-to-end results.

Sources. The previous version of the research presented in this and the next two chapters appeared in [73]. Part of the Sections 4.2, 5.2, and 5.3 previously appeared in the accompanying technical report [74].

4.1 Motivating Example

Figure 4.3 presents a (simplified) implementation of an algorithm that scales an image to a larger size. It consists of the function `scale` and the function `scale_kernel`.

The function `scale` takes as input the scaling factor `f` (which increases the image size in both dimensions), along with integer arrays `src`, which contains the pixels of the image to be scaled, and `dest`, which contains the pixels of the resulting scaled image. The algorithm calculates the value of each pixel in the final result by mapping the pixel's location back to the original source image and then taking a weighted average of the neighboring pixels. The code for `scale` implements the outer portion of the algorithm, which enumerates over the pixels in the destination image.

The function `scale_kernel` implements the core kernel of the scaling algorithm. The algorithm computes the location in the array `src` of the pixel's neighboring four pixels (Lines 3-4), adjusts the locations at the image edges (Lines 6-13), and fetches the pixels (Lines 15-18). To average the pixel values, the algorithm uses *bilinear interpolation.* Bilinear interpolation takes the weighted average of the four neighboring pixel values. The weights are computed as the distance from the source coordinates `i` and `j` to the location of each of the pixels (Lines 20-23). In the last step, the algorithm extracts each RGB color component of the pixel, computes the weighted average, and then returns the result (Lines 25-29).

4.1.1 Reliability Specification

We will use the following reliability specification for this kernel is:

```
int<0.995 * R(i, j, src, s_height, s_width)>
scale_kernel (float i, float j, int[] src, int s_height, int s_width);
```

```
1  int scale_kernel(float i, float j, int[] src, int s_height, int s_width)
2  {
3    int previ = floor(i), nexti = ceil(i);
4    int prevj = floor(j), nextj = ceil(j);
5
6    if (s_height <= nexti) {
7      previ = max(s_height-2, 0);
8      nexti = min(previ+1, s_height-1);
9    }
10   if (s_width <= nextj) {
11     prevj = max(s_width-2, 0);
12     nextj = min(prevj+1, s_width-1);
13   }
14
15   int ul = src[IDX(previ, prevj, s_width)];
16   int ur = src[IDX(nexti, prevj, s_width)];
17   int ll = src[IDX(previ, nextj, s_width)];
18   int lr = src[IDX(nexti, nextj, s_width)];
19
20   float ul_w = (nextj - j) * (nexti - i);
21   float ur_w = (nextj - j) * (i - previ);
22   float ll_w = (j - prevj) * (nexti - i);
23   float lr_w = (j - prevj) * (i - previ);
24
25   int r = (int) (ul_w * R(ul) + ur_w * R(ur) + lr_w * R(lr) + ll_w * R(ll));
26   int g = (int) (ul_w * G(ul) + ur_w * G(ur) + lr_w * G(lr) + ll_w * G(ll));
27   int b = (int) (ul_w * B(ul) + ur_w * B(ur) + lr_w * B(lr) + ll_w * B(ll));
28
29   return COMBINE(r, g, b);
30 }
31
32 void scale(float f, int[] src, int s_width, int s_height,
33                     int[] dest, int d_height, int d_width)
34 {
35   float si = 0, delta = 1 / f;
36
37   for (int i = 0; i < d_height; ++i) {
38     float sj = 0;
39     for (int j = 0; j < d_width; ++j) {
40       dest[IDX(i, j, d_width)] = scale_kernel(si, sj, src, s_height, s_width);
41       sj += delta;
42     }
43     si += delta;
44   }
45 }
```

Figure 4.3: Image Scaling Kernel

The reliability specification of `scale_kernel` appears as part of the type signature of the function. The additional reliability information `0.995 * R(i, j, src, s_height, s_width)` specifies the reliability of the return value:

- **Input Dependencies.** The reliability of the return value is a function of the reliability of the function's inputs. The term `R(i, j, src, s_height, s_width)` represents the *joint reliability* of the inputs on entry to the function, which is the probability that they all together contain the correct result.
- **Reliability Degradation.** The coefficient 0.995 expresses the *reliability degradation* of the function. Specifically, the coefficient is the probability that the return value is correct given that all input variables have the correct values on entry to the function. We describe how to derive this coefficient in Section 4.1.2.

Since the specification does not explicitly state the acceptable absolute difference, it is by default `d` = 0. Therefore, whenever the computation executes without errors, it should produce an exact result.

Arrays. The Rely base language contains annotations on the array parameters that specify that it is allocated in approximate memory. For instance, the following signature of `scale_kernel` would state that the pixel array `src` is in an approximate memory region named `urel`:

```
int<...> scale_kernel (..., int[] src in urel, ...);
```

To generate such annotations, Chisel explores the possibility that the array passed as a `src` parameter may be allocated in the approximate memory. Specifically, Chisel's optimization problem encodes both alternatives, i.e., when `src` is allocated in an exact memory and when it is allocated in an approximate memory. Chisel will report to the developer if this alternative allocation strategy (which may save additional energy) still satisfies the reliability specification. The developer can then annotate the array's allocation statement to indicate to the compiler or the runtime system to allocate the array in an approximate memory.

4.1.2 Obtaining Kernel's Reliability Specification

To obtain a reliability specification for the kernel, a developer typically relates the reliability of the kernel's output to the program's end-to-end sensitivity metric. We present two general strategies for obtaining kernel's reliability degradation, using sensitivity profiling and analytic bound derivation.

(a) 0.20 (b) 0.40 (c) 0.60 (d) 0.80 (e) 0.90 (f) 0.99 (g) 0.999

Figure 4.4: Sensitivity Profiling for Image Scaling for Different Values of `r`

Sensitivity Profiling. Chisel's sensitivity profiler assists the developer in deriving the reliability specification of the kernel. It takes three inputs from the developer:

- **Sensitivity Metric.** A function that compares the outputs of the original and approximate executions. It produces a numerical value that characterizes the difference between the two outputs. For computations that produce images, such as `scale`, a typically used metric is Peak-Signal-to-Noise Ratio (PSNR).

- **Sensitivity Goal.** The value of the sensitivity metric that characterizes the acceptable output quality of the approximate program.

- **Sensitivity Testing Procedure.** A developer can write fault injection wrappers that inject noise into the computation. In general, the developer may use these wrappers to explore the sensitivity of the program's results to various coarse-grained error models. For `scale_kernel`, a developer can implement the following simple sensitivity testing procedure, which returns a random value for each color component:

```
int scale_kernel_with_errors(float i, float j, int[] src, int s_height, int s_width) {
      return COMBINE(rand()%256, rand()%256, rand()%256);
}
```

Chisel's sensitivity profiler automatically explores the relation between the probability of approximate execution and the quality of the resulting image for the set of representative images. Conceptually, the profiler transforms the program to execute the correct implementation of `scale_kernel` with probability `r`, which represents the target reliability. The framework executes the faulty implementation `scale_kernel_with_errors` with probability `1-r`. The framework uses binary search to find the probability `r` that causes the noisy program execution to produce results with acceptable PSNR. The profiler can also plot the quality of the result as a function of `r`.

Figure 4.4 presents a visual depiction of the results of scaling for different values of `r`. Note that implementations with low reliability (0.20-0.80) do not produce acceptable results. However, as `r` reaches values in the range of 0.99 and above, the results become an acceptable approximation of the result of the original (exact) implementation. For the remainder of this section, we use 0.995 as `scale_kernel`'s target reliability, which yields images with an average PSNR of 30.9 dB.

Analytical Lower Bound on Sensitivity Metric. Starting with a reliability specification for our example kernel, it is also possible to obtain an analytic *lower bound* for the sensitivity metric. Specifically, the PSNR for the exact image d and the approximate image d' is

$$\mathrm{PSNR}(d, d') = 20 \cdot \log_{10}(255) - 10 \cdot \log_{10}\left(\frac{1}{3hw}\sum_{i=1}^{h}\sum_{j=1}^{w}\sum_{c\in\{R,G,B\}}(d_{ijc} - d'_{ijc})^2\right).$$

The constants h and w are the height and width of the image and R, G, and B are the color components of a pixel. Each color component is a value between 0 and 255.

The kernel computation computes the value of d'_{ijc} for all three RGB components correctly with probability r. In this case, $\sum_{c\in\{R,G,B\}}(d_{ijc} - d'_{ijc})^2 = 0$. With probability $1 - r$, the kernel computation can compute the value of d'_{ijc} incorrectly. The upper bound on the expected error is then $\sum_{c\in\{R,G,B\}}(d_{ijc} - d'_{ijc})^2 \leq 3 \cdot 255^2$. Therefore, the lower bound on the expected PSNR metric is

$$\mathrm{PSNR}(d, d') \geq -10 \cdot \log_{10}(1 - r).$$

For a reliability specification $r = 0.995$, we can obtain that the expected PSNR is greater than 23.01 dB for any image (and for the typical images used in profiling the average PSNR is greater than 30.9 dB). If, on the other hand, a developer wants to obtain the expected PSNR for any image, then he or she can compute using the previous formula that the reliability degradation should be 0.999.

4.1.3 Optimization Results

Chisel generates expressions that characterize the energy savings and reliability of the function `scale_kernel`. These expressions are parameterized by an unknown *configuration* of the approximate kernel, which specifies which operations and array parameters may be approximate or must be exact. This configuration is the solution to the optimization problem. For

the hardware platforms analyzed in Chapter 6, the optimization algorithm delivers 19.35% energy savings, which is over 95% of the maximum possible energy savings for this computation (which occurs when the reliability bound is zero, and therefore all operations and the `src` and `dest` arrays can be approximate).

When the result of the function is assigned directly to an array variable, like in the case of the `dest` array, the optimization treats this variable (unless specified otherwise by the developer) as another array parameter of the kernel function that can be specified as approximate. Chisel identifies both `src` and `dest` arrays as potentially approximate. Chisel also identifies around 20% of the arithmetic operations as approximate. These operations are in the part of the computation that performs bilinear interpolation. For instance, the assignment to the variable `lr_w` on line 23 uses the inexact multiplication operation "`*.`".

Identifying the kernel's array parameters as approximate informs the developer that the kernel can satisfy its reliability specification with the array allocated in approximate memory. Given this information, the developer can use a predefined API call at the array allocation site to allocate the array in approximate memory across the entire application.

Final Sensitivity Validation. Using the specified sensitivity bound and metric, the framework can evaluate the generated approximate kernel computation on a set of (previously unseen) production inputs. For our example benchmark, the average PSNR on a set of production inputs is 32.31 dB.

4.2 Approximate Hardware Specification and Semantics

The code of `scale` in Section 4.1 illustrates the syntax of the Rely base language, which is a pointer-less C-like language with first-class one-dimensional arrays and reliability specifications. In this section, we present a hardware model and a compilation model for Chisel that captures the basic properties of approximate hardware.

4.2.1 Hardware Specification

We consider a single-CPU architecture model that exposes an ISA with approximation extensions and an approximate memory hierarchy.

Machine Language Syntax

Figure 4.5 presents the abbreviated syntax of the assembly language of the architecture.

$$r \in R \cup \{\text{pc}, \text{bp}\} \qquad n \in \text{Int}_\text{N}$$
$$a \in A \subseteq \text{Int}_\text{N} \qquad \kappa \in K = \{0, 1\}$$

$$op \in Op ::= \texttt{add} \mid \texttt{fadd} \mid \texttt{mul} \mid \texttt{fmul} \mid \texttt{cmp} \mid \texttt{fcmp} \mid \ldots$$
$$\{texttti \in \texttt{I} ::= r = op^\kappa \; r_{oper1} \; r_{oper2} \mid \texttt{jmp} \; r_{condition} \; r_{addr} \mid \texttt{return} \; r_{value} \mid$$
$$r_{value} = \texttt{init} \; n \mid r_{value} = \texttt{load} \; r_{addr} \mid \texttt{store} \; r_{addr} \; r_{value} \mid$$
$$r_{value} = \texttt{loada} \; r_{addr} \; r_{idx} \mid \texttt{storea} \; r_{addr} \; r_{idx} \; r_{value}$$

Figure 4.5: Assembly Language Syntax

Operands. Each operand is either a register $r \in R$ or a fixed N-bit (e.g., 32-bit or 64-bit) integer $n \in \text{Int}_\text{N}$. Floating point numbers are integers coded with the IEEE 754 representation.

Instructions. Each instruction $\texttt{i} \in \texttt{I}$ is either an ALU/FPU arithmetic operation (such as add, multiply and compare), a conditional branch to an address (`jmp`), or a load/store from memory for scalars (`load` and `store`), and for arrays (`loada` and `storea`).

Each arithmetic instruction also has a *kind* $\kappa \in K = \{0, 1\}$, such as $r = \texttt{add}^\kappa \; r_1 \; r_2$, that indicates that the instruction is either exact ($\kappa = 0$) – and always produces the correct result – or approximate ($\kappa = 1$) – and may produce an incorrect result with some probability.

Components of Hardware Specification

The reliability portion of the hardware specification $\psi \in (Op \to \mathbb{R}) \times (\mathbb{R} \times \mathbb{R}) \times (\mathbb{R} \times \mathbb{R})$ is a triple of structures that specify the reliability of instructions, the approximate memory region, and the approximate cache region, respectively. In Sections 5.3.1 and 5.4.1, we extend the hardware specification to include the hardware's accuracy and energy parameters.

Instructions. The projection π_op selects the first element of the hardware specification, which is a finite map from operations to reliabilities. The reliability of an operation is the probability that the operation executes correctly.

Memories. The hardware exposes an exact main memory region and an approximate memory region. The projection π_mem selects the second element of the hardware specifica-

tion, which is a pair (r_{ld}, r_{st}) that denote the reliabilities of loading and storing a value in the approximate main memory.

Caches. The hardware exposes an exact cache and an approximate cache. The projection $\pi_\$$ selects the third element of the hardware specification, which is a pair (r_{ld}, r_{st}) that denote the reliabilities of loading and storing a value in the approximate cache.

4.2.2 Hardware Semantics

Register Files, Memories, Memory Configurations, Programs, and Environments. A *register file* $\sigma \in \Sigma = R \to \mathrm{Int_N}$ is a finite map from registers to machine integers. A *memory* $m \in M = A \to \mathrm{Int_N}$ is a finite map from addresses to machine integers. A memory configuration $\xi \in \Xi = A \to K$, maps an address a to a kind κ that designates whether the memory at a is configured as exact ($\kappa = 0$) or approximate ($\kappa = 1$). A *program* $\gamma \in \Gamma = A \to I$ is a finite map from addresses to instructions. An *environment* $\varepsilon \in \mathrm{E} = \Sigma \times M$ is a register file and memory pair.

Ready Instructions and Configurations. A *ready instruction* $\hat{i} \in \hat{I} := instr \mid \cdot$ is either an instruction $instr \in I$ or the distinguished element "$\cdot$" that indicates that the next instruction needs to be fetched from memory (as determined by the pc register). A *configuration* $\langle \hat{i}, \varepsilon \rangle$ is a ready instruction and environment pair.

Errant Result Distributions. The discrete probability distribution $P_f(n_f \mid op, n_1, n_2)$ models the manifestation of an error during an incorrect execution of an operation. Specifically, it gives the probability that an incorrect execution of an operation op on operands n_1 and n_2 produces a value n_f different from the correct result of the operation.

Arithmetic Instructions. Figure 4.6 presents the inference rules for arithmetic operations. The small-step judgment $\langle \hat{i}, \varepsilon \rangle \xrightarrow[\gamma,\psi,\xi]{\lambda,p} \langle \hat{i}', \varepsilon' \rangle$ denotes that execution of the program γ from the configuration $\langle \hat{i}, \varepsilon \rangle$ under a hardware model ψ and a memory configuration ξ takes a transition with label λ with probability p, yielding a configuration $\langle \hat{i}', \varepsilon' \rangle$.

A *transition label* $\lambda \in \{\mathrm{C}, \langle \mathrm{F}, n \rangle\}$ characterizes whether the transition executed correctly (C) or experienced a fault ($\langle \mathrm{F}, n \rangle$). The value n in a faulty transition records the value that the fault inserted into the semantics of the program. The semantics of an arithmetic operation $r = op^\kappa \ r_1 \ r_2$ takes one of two possibilities:

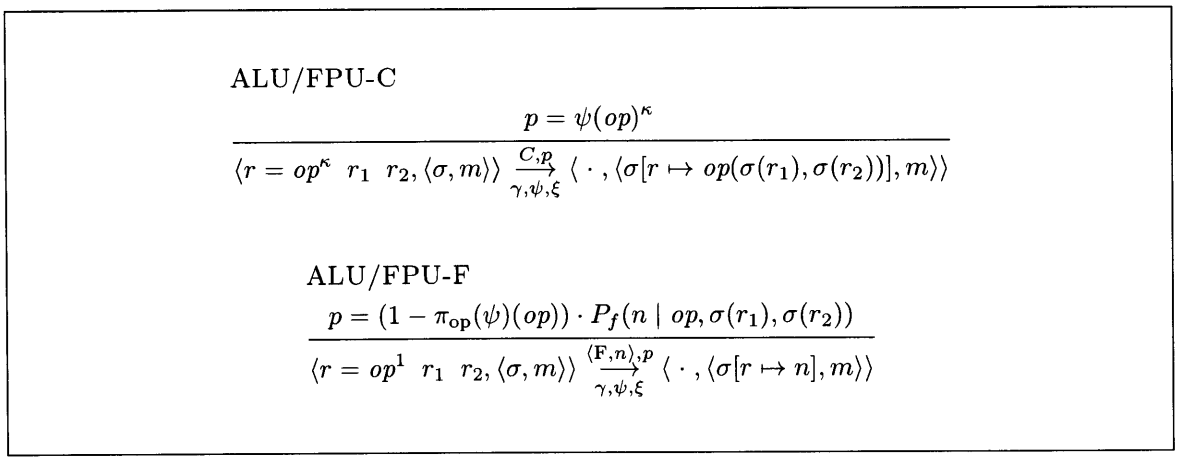


Figure 4.6: Machine Semantics of Arithmetic Operations

- **Correct execution [ALU/FPU-C].** An operation executes correctly with probability $(\pi_{\text{op}}(\psi)(op))^{\kappa}$. We use here the algebraic property that a numerical constant raised to the power zero is equal to 1, and a numerical constant raised to the power one is equal to itself. Therefore, if the operation is exact ($\kappa = 0$) it executes correctly with probability 1. If it is approximate ($\kappa = 1$), then it executes correctly with probability $\pi_{\text{op}}(\psi)(op)$.

  A correct execution proceeds with the rule [ALU/FPU-C] wherein the instruction reads registers r_1 and r_2 from the register file, performs the operation, and then stores the result back in register r.
- **Faulty execution [ALU/FPU-F].** An operation with a kind $\kappa = 1$ experiences a fault with probability $1 - \pi_{\text{op}}(\psi)(op)$. A faulty execution stores into the destination register r a value n that is given by the errant result distribution for the operation, P_f. Note that while the instruction may experience a fault, its faulty execution does not modify any state besides the destination register.

Control Flow. Figure 4.7 presents the semantics of control-flow instructions. These instructions execute reliably. For instance, fetching and decoding instructions is always correct – the machine identifies the instruction and updates the program counter, which is also stored in a fully reliable register.

Control flow transfer instructions, such as `jmp`, always correctly transfer control to the destination address. Preserving the reliability of control flow transfers guarantees that an approximate program always takes paths that exist in the static control flow graph

of the program. We note that while control flow transfers themselves execute correctly, the argument to a control transfer instruction (e.g., the test condition of a `jmp`, which the previous computation stored in the register r_{cond}) may depend on approximate computation. Therefore, an approximate program may, in principle, take a path that differs from that of the original (exact) program.

$$\textsc{Fetch}\quad \frac{i = \gamma(\sigma(\mathrm{pc}))\qquad \sigma' = \sigma[\mathrm{pc} \mapsto \sigma(\mathrm{pc}) + 1]}{\langle \cdot, \langle \sigma, m\rangle\rangle \xrightarrow[\gamma,\psi,\xi]{C,1} \langle i, \langle \sigma', m\rangle\rangle}$$

$$\textsc{Jmp-True}\quad \frac{\sigma(r_{cond}) \neq 0}{\langle \texttt{jmp}\ r_{cond}\ r_{offset}, \langle \sigma, m\rangle\rangle \xrightarrow[\gamma,\psi,\xi]{C,1} \langle \cdot, \langle \sigma[\mathrm{pc} \mapsto \sigma(r_{offset})], m\rangle\rangle}$$

$$\textsc{Jmp-False}\quad \frac{\sigma(r_{cond}) = 0}{\langle \texttt{jmp}\ r_{cond}\ r_{offset}, \langle \sigma, m\rangle\rangle \xrightarrow[\gamma,\psi,\xi]{C,1} \langle \cdot, \langle \sigma, m\rangle\rangle}$$

Figure 4.7: Machine Semantics of Control Flow Instructions

Initialization. Initializing a register with a constant value is always an exact instruction. Therefore, $\langle r = \texttt{init}\ n, \langle \sigma, m\rangle\rangle \xrightarrow[\gamma,\psi,\xi]{C,1} \langle \cdot, \langle \sigma[r \mapsto n], m\rangle\rangle$.

Loads and Stores. Figure 4.8 presents the semantics of the scalar load and store operations. The semantics of loads and stores are similar to arithmetic operation semantics in that each operation can either execute correctly or encounter a fault. The memory configuration ξ determines if an accessed address's memory region that contains the address a is exact (all operations on the region execute correctly) or approximate (operations may encounter a fault).

The function memrel(ψ, a, op, ξ) computes the probability that the operation executed correctly, given the approximate hardware model ψ, address, the operation op, and the memory configuration ξ. When $\xi = 0$ (exact), then memrel$(\psi, a, op, \xi) = 1$ (fully reliable). When $\xi = 1$ (approximate), memrel returns a probability that is less than 1, and depends on the reliability of the main memory and the cache memory. We discuss how to compute memrel below, when discussing handling of caches.

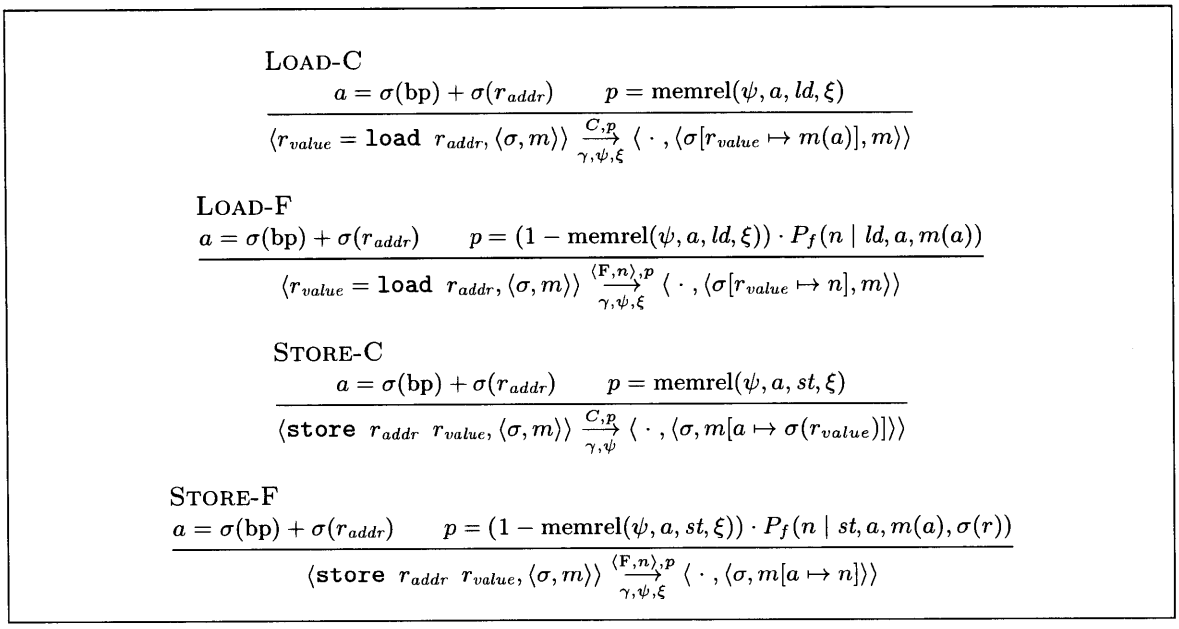


Figure 4.8: Machine Semantics of Loads and Stores

As with the destination register of arithmetic operations, if a load or store instruction encounters a fault, then only the contents of the destination register or memory address are modified. The addresses of offsets (located in the base pointer register bp) for the load and store instructions are performed exactly.

Array Loads/Stores. Figure 4.9 presents the semantics of array load and store operations. The semantics of loads and stores of arrays is failure oblivious [97] in that an out-of-bounds array access never forces the program to halt. These checks are implemented differently for loads and stores.

- **Loads.** Array loads do not include an explicit bounds check. The rule [LOAD-ARR1] uses the pointer to the array's data $\sigma(r_{arr})$ and the index value passed in as an index (r_{idx}) to compute the address of the data (a). If that address is a valid memory location ($a \in \text{dom}(m)$) then the rule loads the value of the address with probability $\text{memrel}(\psi, a, ld, \xi)$. Note that this location can still be outside the array bounds (if the previous approximate execution computed the index inexactly). The rule [LOAD-ARR2] states that in case when a is not a valid memory address ($a \notin \text{dom}(m)$), the semantics is free to place any value n into the destination register r. This semantic approach minimizes the overhead incurred on array reads by only requiring an implementation to check if the address is valid.

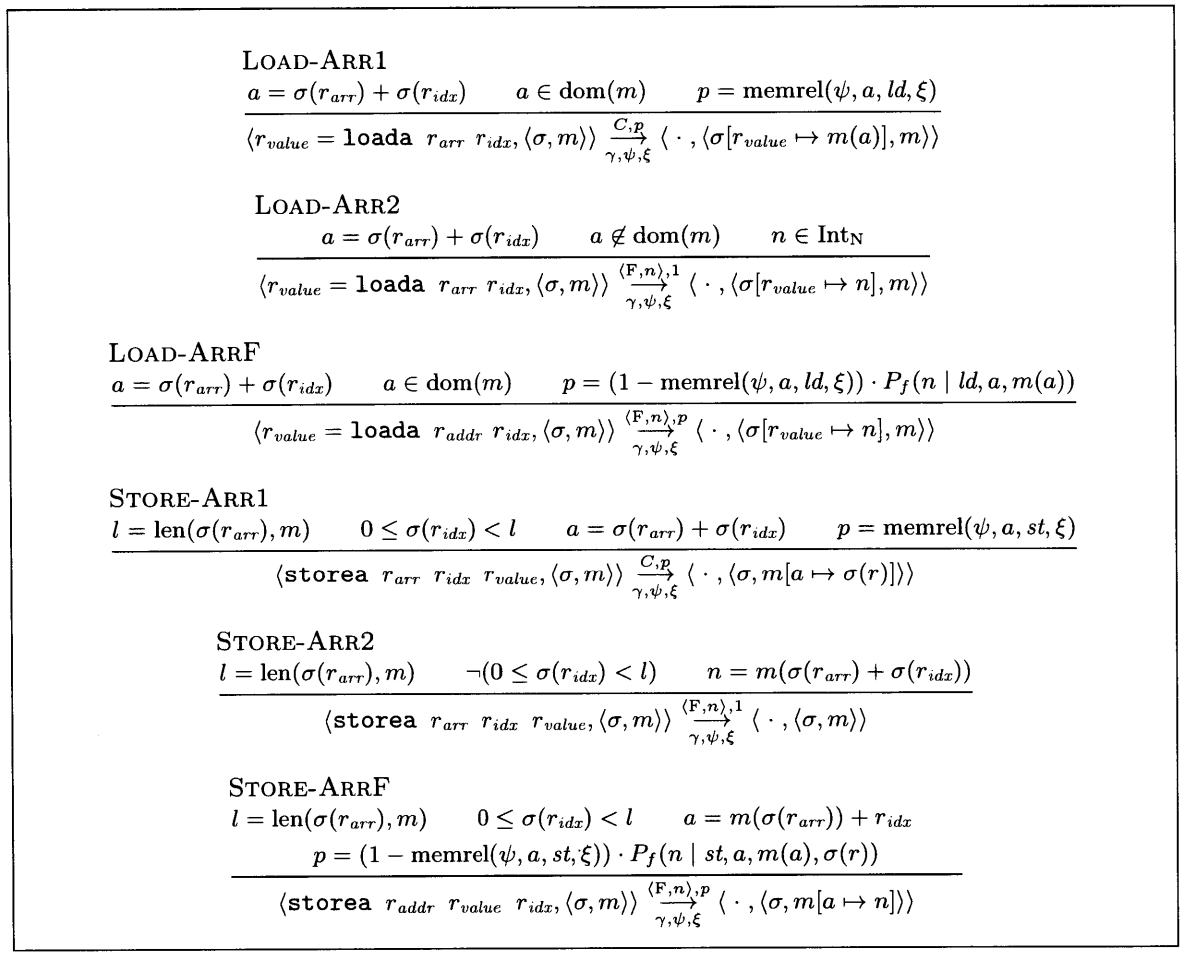


Figure 4.9: Machine Semantics of Array Loads and Stores

- **Stores.** Array stores require an array bounds check so that our analysis can perform modular reasoning in the presence of errant array indices. Specifically, the semantics of the store operations prohibits writing to locations outside of the array boundaries (rule [STORE-ARR-1]). The function len obtains the length of an array (we discuss the representation of arrays later in this section). If the instruction attempts to store a value to an out-of-bound address, such updates are ignored (rule STORE-ARR2).

The rules for the cases when the read from memory and write to memory fail are analogous to the rules for the loads and stores of scalar variables.

Arrays and Caches. To precisely assign the reliabilities of memory operations, Chisel treats the cache memory as an additional buffer between the CPU and the main memory

in the approximate hardware model. While the semantics does not represent the cache memory explicitly, it accounts for the cache by assigning the reliability of the transitions. Below, we describe how to assign those transition probabilities.

When reading data using the `load` instruction, the processor checks whether the cache contains the data's address. If it does, then the processor reads the data from the cache. The reliability in this case is bounded by the probability that 1) the initial read of the data from the main memory and write to the cache were successful and 2) the read of the data from the cache was successful. If the data is not in the cache, the processor reads the data from the main memory. The reliability of read is then also bounded by the probability that 1) reading the from the main memory, and 2) writing and reading the data from the cache succeeded. The reliability of `load` is then the minimum of the two cases. Therefore, if $\xi(a) = 1$ (data is in approximate memory), then $\text{memrel}(\psi, a, ld, \xi) = \pi_1(\pi_{\text{mem}}(\psi)) \cdot \pi_1(\pi_{\$}(\psi)) \cdot \pi_2(\pi_{\$}(\psi))$.

When writing data using the `store` instruction, the processor implements a write-through policy – if the write-to address is already in the cache, the data is written both to the cache and the main memory. The reliability of the write operation in this case is the probability that 1) writing the data to the cache was successful and 2) writing the data to the main memory was successful. However, note that the probability of writing to the cache matters only if the data is subsequently read from the cache (instead of the main memory). But since the conservative analysis of the reliability of `load` already assumes that the data is not in the cache when assigning reliability for the subsequent reads, we can drop this term. If the write-to address was not in the cache, then the cache implements the no-write-allocate policy – the processor writes the data directly to the main memory. The reliability in this case is equal to the probability that the write to memory succeeded. The reliability of `store` is then equal to the probability of successfully writing the data to the main memory. Therefore, if $\xi(a) = 1$ (data is in approximate memory), then $\text{memrel}(\psi, a, ld, \xi) = \pi_2(\pi_{\text{mem}}(\psi))$.

4.2.3 Compilation and Runtime Model

Data Allocation. The compilation and runtime system allocates the data in memory as follows. The program's instructions and the stack are stored in the exact memory region. The system represents arrays with a header and a separately allocated chunk of memory that contains the array's data. The header contains the length of the array's data and the address of the array's data in memory. The system allocates the header in exact main memory and allocates the data chunk in either exact or approximate memory based upon Chisel's optimization results. This allocation strategy enables the system to separate the reliability of the array's metadata from the reliability of the data stored in the array.

To support our formalization of reliability, we define a *variable allocation* $v \in V \to \mathcal{P}(A)$ as a finite map from a *program variable* $v \in V$ to the address (or set of addresses in the case of an array) in memory at which the variable has been allocated by the compilation and runtime system.

Data and Cache Memory. During the execution, the system stores data allocated in the exact region of the main memory in the exact cache. The system can store data allocated in the approximate main memory in either the exact or approximate cache. If the data is initialized in the outer implementation language (and used in possibly multiple kernels), a developer can, based on the result of Chisel's optimization, annotate the allocation site of the array variable in the code written in the implementation language.

Calling Convention. We adopt the standard C calling convention for the function with the Chisel instructions. The kernel function has only a single return statement.

4.2.4 Big-step Semantics

We use the following big-step semantics to define a program's reliability and accuracy.

Definition 1 (Big-step Trace Semantics).

$$\langle \cdot, \varepsilon \rangle \underset{\gamma,\psi,\xi}{\Longrightarrow} (\varepsilon', \tau, p) \equiv \langle \cdot, \varepsilon \rangle \xrightarrow[\gamma,\psi,\xi]{\lambda_1, p_1} \cdots \xrightarrow[\gamma,\psi,\xi]{\lambda_n, p_n} \langle \cdot, \varepsilon' \rangle$$
$$\textit{where } \tau = \lambda_1, \ldots, \lambda_n,\ p = \prod_{i=1}^{n} p_i \textit{ and } \textit{final}(\langle \cdot, \varepsilon' \rangle, \gamma)$$

The big-step trace semantics is a reflexive transitive closure of the small-step execution relation that records a trace of the program's execution. A *trace* $\tau \in \mathrm{T} ::= \cdot \mid \lambda :: \mathrm{T}$ is a sequence of small-step transition labels. The *probability of a trace*, p, is the product of the probabilities of each transition. The predicate *final* $\subseteq (I \times \mathrm{E}) \times \Gamma$ indicates that the program cannot make a transition from the configuration.

Definition 2 (Big-step Aggregate Semantics).

$$\langle \cdot, \varepsilon \rangle \underset{\gamma,\psi,\xi}{\Longrightarrow^*} (\varepsilon', p) \textit{ where } p = \sum_{\tau \in \mathrm{T}} p_\tau \textit{ such that } \langle \cdot, \varepsilon \rangle \underset{\gamma,\psi,\xi}{\Longrightarrow} (\varepsilon', \tau, p_\tau)$$

The big-step aggregate semantics enumerates over the set of all finite length traces and sums the aggregate probability that a program γ starts in an environment ε and terminates in an environment ε' given a hardware specification ψ and memory configuration ξ.

5 Chisel Optimization Algorithm

This chapter presents Chisel's optimization algorithm, which uses program analysis to reduce the problem of selecting approximate operations and variables to an integer linear optimization problem. We first outline the main components of the algorithm.

Integer Linear Programming (ILP). A general ILP problem has the form:

$$
\begin{aligned}
&\textbf{\textit{Minimize:}} && \sum_{i=1}^{n} c_i \cdot x_i \\
&\textbf{\textit{Constraints:}} && \sum_{i=1}^{n} a_{1,i} \cdot x_i \leq b_1 \\
& && \cdots \\
& && \sum_{i=1}^{n} a_{m,i} \cdot x_i \leq b_m \\
&\textbf{\textit{Variables:}} && x_1, \ldots, x_n \in \{0, 1\}
\end{aligned}
$$

The variables $x_1, \ldots, x_n$ are called *decision variables.* Each of these variables can take a value of 0 or 1. The coefficients $a_{1,1}, \ldots, a_{m,n}$, $b_1, \ldots, b_m$, and $c_1, \ldots, c_n$ are real-valued numerical constants. The function that is being minimized is called the *optimization objective.* The *optimization constraints* are the inequalities that bound the sums of decision variables. The objective and the constraints are *linear* functions of the decision variables.

Solving integer linear optimization problems is a well-studied problem in many areas of science, mathematics, and engineering. An ILP solver takes as input a problem with the exposed objective, constraints, and the decision variables. The solver finds the assignment of the variables $x_1, \ldots, x_n$ that minimizes the objective, while satisfying all constraints. Although solving ILP problems is in general computationally hard, modern solvers (such as Gurobi [48]) can find the optimal solution even of large ILP problems or a good approximation, which satisfies the constraints but does not necessarily minimize the objective.

To reduce the placement of approximate operations and storing data in approximate memory to an integer linear problem, Chisel performs the following tasks:

Specify Decision Variables. Chisel represents the choice whether to approximate each arithmetic operation or a variable in a kernel as one decision variable. Recall that the instructions in Chisel's language have a kind (which specifies that an instruction can be either exact or approximate). Each such kind is a decision variable in the optimization problem. To support this formalization, Section 5.1 presents the notion of *labels* that uniquely identify instructions and variables, and the *kind configuration*, which maps the labels to the kinds of instructions or variables.

Compute Reliability and Accuracy Constraints. Chisel generates reliability and accuracy constraints via a static *precondition generator analysis*. In general, a precondition generator C_ψ operates backward and is often used in program verification, and recently in program synthesis [116]. It takes as input a predicate Q_{post} and the program statements $\langle S_1, \ldots, S_n \rangle$. It produces a predicate $Q_{pre} = C_\psi(\langle S_1, \ldots, S_n \rangle, Q_{post})$, such that if Q_{pre} is valid before the execution of the kernel, then Q_{post} will be valid at the end of the execution.

The predicates Q_{post} for the kernel come from the function specifications, which can state that, e.g., the reliability of the kernel's result should be greater than `0.99*R(x)`. The analysis starts by constructing the corresponding predicate $Q_{post} :=$ `0.99*R(x)` $\leq$ `R(result)`. The analysis transforms each such predicate, operating backward to produce Q_{pre} by analyzing the statements from S_n to S_1 recursively, i.e., $Q_{pre} := C_\psi(\langle S_1, \ldots, S_{n-1} \rangle, C_\psi(\langle S_n \rangle, Q_{post}))$.

Sections 5.2 and 5.3 present Chisel's reliability and accuracy constraint generators. The analysis produces predicates Q_{pre}, which are parameterized by the instruction and variable labels. For any particular set of labels, the predicates Q_{pre} are valid for *all* inputs specified in the kernel's input specification.

To find the assignments of labels that maximize savings, Chisel's optimization algorithm first solves the part of the predicates that does not depend on the labels. Chisel then transforms the remaining terms of the predicates into the ILP optimization constraints, by substituting the labels with the corresponding instruction/variable kinds (ILP problem's decision variables) and transforming the predicates to be linear functions of the kinds.

Compute Energy Savings Objective. Chisel generates the energy savings objective using a dynamic program analysis. Section 5.4 presents how Chisel constructs an estimate of the savings (as the function of the decision variables) from the traces of the kernel when executed on representative inputs.

Generate Optimized Kernel. Chisel dispatches the generated ILP problem to an off-the-shelf solver, which returns the kinds of instructions and variables that maximize savings. Chisel uses these kinds to generate the kernel with approximate and exact operations.

5.1 Configurable Approximate Programs

To enable optimization with approximate instructions and data, we augment our program representation to create an intermediate representation that includes *labels*, where each label $\ell \in \mathcal{L}$ is uniquely associated with an instruction or a program variable. Labels enable Chisel to separately mark each instruction and variable as either exact or approximate.

5.1.1 Labeled Instructions and Variables

Instructions. We augment each arithmetic instruction to have a label instead of a kind:

$$i \in I ::= \; r = op^{\ell} \; r \; r$$

Program Variables. We define the finite map $\chi \in V \to \mathcal{L}$ that maps each variable in the program to a unique label.

Kind Configurations. We define a *kind configuration* $\theta \in \Theta = \mathcal{L} \to K$ as a finite map from labels to kinds that denotes a selection of the kind (i.e., exact or precise) of each of the program's instructions and variables. The set of kind configurations denotes that set of all possible optimized programs that Chisel can generate. Any element of this set represents one approximate version of the program. We also make the following auxiliary definitions:

- **Approximated program $\gamma[\theta]$.** Substitution $\gamma[\theta]$ represents the program generated by substituting each label ℓ in the program γ by the corresponding kind $\theta(\ell)$.
- **Kinded memory configuration map ξ_θ.** For each variable $v \in V$ in the program γ, it labels all addresses $a \in \upsilon(v)$ with the variable's kind, i.e., $\xi(a) = \theta(\chi(v))$.

5.1.2 Intermediate Language for Analysis

Chisel translates the statements from Rely to an intermediate language which maintains information about structured control flow. Figure 5.1 presents the syntax of the expressions and statements in the analyzable part of the Rely source language. It operates on numerical or boolean data. It supports scalar integer and floating point scalar variables and multidimensional arrays. The variables can be stored in the exact or approximate memories. The control flow includes conditional branching and finite loops.

Figure 5.2 presents the syntax of the intermediate language. A statement in this language can be a labeled assembly instruction (except a `jmp` instruction), a sequence of statements,

$$
\begin{array}{lcl}
n \in \text{Int}_N & e \in \mathit{Exp} & ::= \; n \mid x \mid a[\mathit{Exp}^+] \mid \texttt{(}\mathit{Exp}\texttt{)} \mid \mathit{Exp}\ \mathit{iop}\ \mathit{Exp} \mid \mathit{Exp}\ \mathit{fop}\ \mathit{Exp} \\
x \in \text{Var} & b \in \mathit{BExp} & ::= \; \texttt{true} \mid \texttt{false} \mid \mathit{Exp}\ \mathit{cmp}\ \mathit{Exp} \mid \texttt{(}\mathit{BExp}\texttt{)} \mid \\
a \in \text{ArrVar} & & \quad \mathit{BExp}\ \mathit{lop}\ \mathit{BExp} \mid \texttt{!}\mathit{BExp}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Kernel} & ::= & \mathit{Decl}^*;\ \mathit{Stmt};\ \texttt{return}\ \mathit{Exp} \\
\mathit{Decl} & ::= & [\,\texttt{int} \mid \texttt{float} \mid \texttt{int[}n^+\texttt{]} \mid \texttt{float[}n^+\texttt{]}\,]\ x \\
\mathit{Stmt} & ::= & \texttt{skip} \mid x\ \texttt{=}\ \mathit{Exp} \mid a\texttt{[}\mathit{Exp}^+\texttt{]}\ \texttt{=}\ \mathit{Exp} \mid \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt} \mid \\
 & & \texttt{if}\ \mathit{BExp}\ \mathit{Stmt}\ \mathit{Stmt} \mid \texttt{while}\ \mathit{BExp}\ \texttt{:}\ n\ \mathit{Stmt}
\end{array}
$$

Figure 5.1: Syntax of the Analyzable Part of the Rely Language [19]

$$
s \in S \;:=\; i \;\mid\; s;s \;\mid\; \texttt{if}\ r_{cond}\ s_{then}\ s_{else}
$$

Figure 5.2: Chisel's Intermediate Language

or an intermediate conditional statement, which, based on the result of the register r_{cond}, continues the execution of the statements in the *then* or *else* branches.

The translation of the statements from Rely to the intermediate language is straightforward. Rely's assignment statements are translated to the sequence of arithmetic and memory assembly instructions. The translation assumes a compilation model with unbounded number of pseudo-registers, where the destination and operand registers do not alias. Translation of conditionals stores the result of the condition expression to the register r_{cond}. Translation of bounded while loops (that execute up to n iterations) unrolls the loop to a sequence of conditionals. Chisel's analysis does not support unbounded while loops.

This translation therefore ensures that the intermediate language has only a structured control-flow and no stray `jmp` instructions. The intermediate representation of the kernel allows a compiler to perform a number of standard instruction-level optimizations (e.g., peephole optimizations, dead code elimination, or invariant code motion) before running the reliability and accuracy analyses. At the same time, it allows an easier presentation of the analysis.

The translation from the intermediate language to the assembly language (Figure 4.5) is also straightforward. The conditionals are implemented using jump instructions. The pseudo-registers are translated to the machine registers using standard mapping algorithms. This translation does not affect the reliability and accuracy analyses, because all registers and the stack memory are exact and jump instructions also execute fully reliably.

5.2 Reliability Constraint Construction

In this section, we adapt the reliability definitions from [19] for Chisel's configurable assembly language kernels. Later, in Section 5.3, we extend the definitions of predicates to represent *combined* accuracy and reliability constraints.

5.2.1 Reliability Predicates

Chisel's generated preconditions are *reliability predicates* that characterize the reliability of an approximate program. A reliability predicate P has the following form:

$$P := R_f \leq R_f \mid P \wedge P \mid \text{True} \mid \text{False}$$
$$R_f := \rho \mid \rho^{\ell} \mid \mathcal{R}(O) \mid R_f \cdot R_f$$

Specifically, a predicate is either a conjunction of predicates or a comparison between *reliability factors*, R_f. A reliability factor has multiple forms:

- **Constant.** A real number, $\rho \in [0, 1]$, represents the probability that an approximate instruction produces a correct result. The analyses will calculate the constants ρ using the elements of the approximate hardware specification ψ.

- **Kinded reliability.** A term ρ^{ℓ}, represents the reliability of an λ-labeled instruction that can be either exact or approximate.

  The label ℓ encodes the choice between the exact and approximate version of an instruction: if $\theta(\ell) = 0$ (exact), this term will have the value 1 (instructions always produce a correct result); if $\theta(\ell) = 1$ (approximate), the term will have the value ρ. We remark the intentional notational similarity of ρ^{ℓ} with numerical exponentiation.

- **Joint Reliability Factor.** A term $\mathcal{R}(O)$ represents probability that all registers and variables in the set $O \subseteq R \cup V$ have the same value in the exact and the approximate executions. This term abstracts the probability that the approximate execution's environment has exact values of the operands from which the remaining execution can produce the exact result.

- **Product of Reliability Factors.** This term combines the probability that the instructions produce correct results and the initial program environments have the exact value of the operands.

This definition of reliability predicates is sufficient for expressing properties about *error frequency*. We will demonstrate their use in this section. We will present the extension of the predicates to support reasoning about *both frequency and magnitude* of error in Section 5.3.

Before we present the formal semantics of the predicates, we present examples that illustrate the intuition behind how we intend to use these predicates in the analysis:

***Example* 1** (Kinded Reliability Factor). *Consider the predicate* $0.9 \leq 0.99^{\ell}$ *with kinded reliability factor. Given a kind configuration* θ*, we can represent this predicate as* $0.9 \leq 1$ *(if* $\theta(\ell) = 0$*) or* $0.9 \leq 0.99$ *(if* $\theta(\ell) = 1$*). We can succinctly rewrite these two predicates over reals as* $0.9 \leq 0.99^{\theta(\ell)}$.

***Example* 2** (Joint Reliability Factor). *A predicate* $0.9 \leq \mathcal{R}(\{x\})$ *bounds the probability that the variable* x *in the approximate execution has the same value as in the exact execution (at the same program point). Chisel's analysis calculates a lower bound on this probability, which is much easier to compute instead of the exact probability* $\mathcal{R}(\{x\})$.

5.2.2 Semantics of Reliability Predicates

Approximate Environment Distribution φ. An approximate environment distribution $\varphi \in \Phi = \mathrm{E} \to \mathbb{R}$ is a probability mass function over possible approximate environments. Specifically, for each approximate environment ε_a, the value $\varphi(\varepsilon_a)$ is the probability that the program's execution is in ε_a. This distribution helps formalize the predicate semantics.

Computing the approximate environment distribution is, in general, intractable and the analyses presented in this dissertation will not explicitly compute this distribution. However, a key idea behind the analysis is to compute a lower bound on the probability that the approximate execution is in the same environment as the exact distribution.

Semantics of Reliability Predicates. The denotation of a reliability predicate, P, $[\![P]\!] \in \mathcal{P}(\mathrm{E} \times \Phi \times \Theta \times \Upsilon)$ is the set of quadruples (exact environment, approximate environment distribution, kind configuration, and variable allocation) that satisfy the predicate.

The denotation of a reliability factor $[\![R_f]\!]$ is the real-valued reliability that results from evaluating the factor for a given quadruple. Figure 5.3 presents the semantics of the reliability factors. Note that the denotation of the kinded reliability factors, ρ^{ℓ}, instantiates the value of the label ℓ's kind, $\theta(\ell) \in \{0, 1\}$. Therefore, it follows from the semantics that if $\theta(\ell) = 0$, then $[\![\rho^{\ell}]\!](\varepsilon, \varphi, \theta, \upsilon) = \rho^0 = 1$, and if $\theta(\ell) = 1$, then $[\![\rho^{\ell}]\!](\varepsilon, \varphi, \theta, \upsilon) = \rho^1 = \rho$. This semantics is analogous to the semantics of arithmetic operations (Figure 2).

$$\boxed{[\![R_f]\!] \in \mathrm{E} \times \Phi \times \Theta \times \Upsilon \to \mathbb{R}}$$

$$[\![\rho]\!](\varepsilon, \varphi, \theta, \upsilon) = \rho \qquad [\![\rho^{\ell}]\!](\varepsilon, \varphi, \theta, \upsilon) = \rho^{\theta(\ell)}$$

$$[\![R_{f1} \cdot R_{f2}]\!](\varepsilon, \varphi, \theta, \upsilon) = [\![R_{f1}]\!](\varepsilon, \varphi, \theta, \upsilon) \cdot [\![R_{f2}]\!](\varepsilon, \varphi, \theta, \upsilon)$$

Figure 5.3: Semantics of Reliability Factors (See Definition 3 for Semantics of Joint Reliability Factors)

The denotation of $\mathcal{R}(O)$ is the probability that an approximate environment ε_a sampled from φ has the same value for all operands $o \in O$ as the exact environment ε.

Definition 3 (Joint Reliability Factor).

$$[\![\mathcal{R}(O)]\!](\varepsilon, \varphi, \theta, \upsilon) = \sum_{\varepsilon_a \in \mathcal{E}(\varepsilon, O, \upsilon)} \varphi(\varepsilon_a),$$

where $\varepsilon \equiv (\sigma, m)$ and $\varepsilon_a \equiv (\sigma_a, m_a)$

and $\mathcal{E}(\varepsilon, O, \upsilon) = \{(\sigma_a, m_a) \mid (\sigma_a, m_a) \in \mathrm{E} \ \wedge$

$\forall o \in O.\ o \in R \Rightarrow \sigma_a(o) = \sigma(o) \ \wedge$

$o \in V \Rightarrow \forall a \in \upsilon(o).\ m_a(a) = m(a)\}.$

The result of the function $\mathcal{E}(\varepsilon, O, \upsilon)$ is the set of approximate environments $\varepsilon_a \equiv (\sigma_a, m_a)$ in which all operands $o \in O$ have the same values as in $\varepsilon \equiv (\sigma, m)$. Specifically, if the operand o is a register location ($o \in R$), then it compares the values in the register files ($\sigma_a(o)$ and $\sigma(o)$). If the operand o is a variable in memory ($o \in V$), then it compares the values in memories (m_a and m) for all addresses $\upsilon(o)$ allocated for this variable. As a special case, $[\![\mathcal{R}(\varnothing)]\!](\varepsilon, \varphi, \theta, \upsilon) = 1$.

5.2.3 Paired Execution Semantics

Given the semantics of reliability predicates, we define the approximate program reliability with a Hoare-triple-like axiomatic semantics for the program's *paired execution semantics*:

Definition 4 (Paired Execution Semantics).

$$\langle \cdot, \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\gamma,\psi}^{\theta,\chi,\upsilon} \langle \varepsilon', \varphi' \rangle \text{ such that } \langle \cdot, \varepsilon \rangle \underset{\gamma[0_\theta],\psi,0_\xi}{\Longrightarrow} (\varepsilon', \tau, 1)$$

$$\text{and } \varphi'(\varepsilon'_a) = \sum_{\varepsilon_a \in \mathrm{E}} \varphi(\varepsilon_a) \cdot p_a$$

$$\text{where } \langle \cdot, \varepsilon_a \rangle \underset{\gamma[\theta],\psi,\xi_\theta}{\Longrightarrow}^{*} (\varepsilon'_a, p_a)$$

$$\text{and } \forall v \in V \;.\; \forall a \in \upsilon(v) \;.\; \xi_\theta(a) = \theta(\chi(v)).$$

The paired execution semantics relates a program's exact execution with its approximate executions via the tuple $\langle \varepsilon, \varphi \rangle$. Here, ε denotes the environment of the program in the exact execution and $\varphi(\cdot)$ is the distribution of approximate environments that correspond to ε. Likewise, $\varphi'(\cdot)$ is the distribution of approximate environments that correspond to ε'. The kind configuration map (θ), a map from variable names to allocation sites (υ), a map from variable names to labels (χ), and the memory configuration map (ξ) are parameters known at the beginning of the execution of γ.

We will now focus on the parts of this formalization:

- **Exact Execution.** The semantics specifies the program's exact execution via a big-step semantics (Definition 1) that starts from the environment ε and ends in ε'. It uses the *exact kind and memory configurations* 0_θ and 0_ξ that specify that all registers and memory locations are reliable.
- **Approximate Execution.** Because approximate operations can produce different results with some probability, the natural representation for the environments of a program's approximate execution is a probability distribution that specifies the probability that the execution is in a particular environment. The semantics specifies the distributions φ and φ' of the approximate execution's initial and final environments ε_a and ε'_a, respectively.

  Also note that the approximate program and memory $\gamma[\theta]$ and ξ_θ are parameterized by the approximate kind configuration θ. Specifically, the program $\gamma[\theta]$ is obtained by substituting the kind labels with the constants in θ and memory configuration ξ_θ contains the appropriate label from θ for each address (see Section 5.1).
- **Approximate Environment Distributions.** The relationship between the two distributions φ and φ' is given by a summation over the approximate traces (see

Definition 2). Specifically, for a final approximate environment ε'_a, the probability $\varphi'(\varepsilon'_a)$ is equal to the sum of terms where each represents the transition from some initial state ε_a (with probability $\varphi(\varepsilon_a)$) to the environment ε'_a (the transitions have the cumulative probability p_a).

Reliability Transformer. Reliability predicates and the semantics of approximate programs are connected through the view of a program as a *reliability transformer*. This definition also extends the original definition from [19]. The definition is similar to the standard Hoare triple relation[1] [38]. It states that if an environment and distribution pair $\langle \varepsilon, \varphi \rangle$ satisfy a reliability predicate P, then the program's paired execution transforms the pair to a new pair $\langle \varepsilon', \varphi' \rangle$ that satisfy a predicate Q:

Definition 5 (Reliability Transformer Relation).

$$\begin{aligned} \psi, \theta, \chi \models \{P\}\ \gamma\ \{Q\} \quad \equiv \quad & \forall \varepsilon, \varphi, \upsilon.\ (\varepsilon, \varphi, \theta, \upsilon) \in [\![P]\!] \Rightarrow \\ & \forall \varepsilon', \varphi'.\ \langle\ \cdot\ , \langle \varepsilon, \varphi \rangle \rangle \Downarrow_{\gamma,\psi}^{\theta,\chi,\upsilon} \langle \varepsilon', \varphi' \rangle \Rightarrow \\ & (\varepsilon', \varphi', \theta, \upsilon) \in [\![Q]\!] \end{aligned}$$

5.2.4 Reliability Precondition Generator

Chisel's reliability constraint generator operates as a precondition generator. Given a postcondition predicate, Chisel's reliability precondition generator produces a precondition that, when true before the execution of the program, ensures that the postcondition is true after. In other words, the precondition, program, and postcondition satisfy the reliability transformer relation we defined in the previous section. The reliability precondition generator is a function $C \in S \times P \to P$ that takes as inputs an instruction and a postcondition and produces a precondition as output.

Initial Postcondition

The analysis starts from the return instruction (the last instruction in the Chisel's kernel). Recall that the specification of reliability of the function's output has the form `int <rspec*R(v1,...,vn)> f(int v1, ..., int vn)`, where `rspec` is the numerical constant

[1]The standard Hoare triple relation $\{P\}\ \gamma\ \{Q\}$ states that if the program's environment satisfies the logical predicate P before executing the program γ, then the environment that is obtained by executing the program γ will satisfy the logical predicate Q.

and `v1,...,vn` are the function's parameters. The analysis represents this specification as the reliability factor $\rho_{spec} \cdot \mathcal{R}(V_{spec})$, where $V_{spec} \subseteq \{v_1, \ldots, v_n\}$ is the set that contains the function inputs in the specification's reliability factor.

The analysis of the return instruction starts from the default initial postcondition $Q_0 = \text{True}$ and constructs the following precondition:

$$C_\psi(\texttt{return } r_{ret}, Q_0) = \rho_{spec} \cdot \mathcal{R}(V_{spec}) \leq \mathcal{R}(\{r_{ret}\}) \wedge Q_0$$

This predicate states that the probability that the return register r_{ret} contains the correct output at the end of the kernel function is greater than the probability that the inputs of the function had correct values at the beginning of the kernel execution, multiplied by the constant ρ_{spec} (which represents the reliability degradation).

If the function has additional output reliability specifications for the array parameters, such as `f(...,int[] <ri*R(v1,...,vn)> vi, ...)`, then the analysis of the return statements starts with the initial postcondition Q_0 that is the conjunction of the predicates of the form $\rho_{spec,i} \cdot \mathcal{R}(V_{spec,i}) \leq \mathcal{R}(\{v_i\})$ for each array parameter v_i that has output specification. This postcondition states that all input array variables have the reliability at least that specified in the function's definition.

***Example* 3** (Analysis of a Function Returning Constant). *We will analyze the function returning a constant written in the Rely language:* `int<0.99> three() { return 3; }` *The compiler generates the two assembly instructions* `r0 = init 3; return r0;`.
The analysis constructs the following precondition for the return instruction: $0.99 \cdot \mathcal{R}(\varnothing) \leq \mathcal{R}(\{r_0\})$. *The left side of the inequality comes from the function specification (and* $\varnothing$ *indicates an empty set, i.e., no variables in the specifications). The right-hand side of the inequality is the result of the rule* $C_\psi(\texttt{return } r_0, \textit{True})$.

Precondition Generator for Statements

Initialization and Sequence. The following equations present the rules for initializing a register with a constant and a sequence of instructions:

$$C_\psi(r = \texttt{init } n, Q) = Q\,[\mathcal{R}(X)/\mathcal{R}(\{r\} \cup X)]$$
$$C_\psi(\texttt{inst1; inst2}, Q) = C_\psi(\texttt{inst1}, C_\psi(\texttt{inst2}, Q))$$

The initialization rule removes the occurrence of the register r in the joint reliability factor because its previous value is not relevant for the reliability of the kernel's outputs.

Specifically, the substitution $Q\,[\,\mathcal{R}(X)/\mathcal{R}(\{r\} \cup X)\,]$ matches all occurrences of the destination register r in a reliability term that occur in the predicate Q and removes them (by leaving only the remainder set X).

The sequence rule first computes the precondition for the second instruction (`inst2`) and passes it as the postcondition to the analysis of the first instruction (`inst1`).

***Example* 4** (Analysis of a Function Returning Constant). *We return to the function from Example 3 and analyze the the instruction* `r1 = init 3;` *and the postcondition* $0.99 \cdot \mathcal{R}(\varnothing) \leq \mathcal{R}(\{r_1\})$ *(that the analysis of the return instruction produced). Then, the precondition generator uses the rules for sequence and initialization to generate the function precondition* $0.99 \cdot \mathcal{R}(\varnothing) \leq \mathcal{R}(\varnothing)$.

ALU/FPU. The following equation presents the generator rule for ALU/FPU operations:

$$C_\psi(r = op^\ell\ r_1\ r_2, Q) = Q\,[\,\rho^\ell_{op} \cdot \mathcal{R}(\{r_1, r_2\} \cup X))/\mathcal{R}(\{r\} \cup X)\,]$$

The rule works by substituting the reliability of the destination register r with the reliability of its operands and the reliability of the operation itself. The substitution $Q[\mathcal{R}(\{r_1, r_2\} \cup X)/\mathcal{R}(\{r\} \cup X)]$ matches all occurrences of the destination register r in a reliability factor inside the predicate Q and replaces them with the input registers, r_1 and r_2. The substitution also multiplies in the factor ρ^ℓ_{op}, which is the reliability of the operation op as a function of its label's kind configuration, where $\rho_{op} = \pi_{\text{op}}(\psi)(op)$.

***Example* 5** (Analysis of Addition). *We analyze the statement* $r = \texttt{add}^\ell r_1, r_2$, *with the postcondition* $Q := 0.99 \leq \mathcal{R}(\{r, z\})$ *and the hardware configuration* ψ.
First, the analysis obtains the reliability of the addition operator $\rho = \pi_{op}(\psi)(add)$. *Second, the analysis uses the instruction's label* ℓ *to represent reliability choice,* ρ^ℓ. *Third, the analysis generates the new reliability factor* $\mathcal{R}(\{r_1, r_2, z\})$ *by substituting* r *with* $\{r_1, r_2\}$. *Finally, the analysis substitutes* $\mathcal{R}(\{r, z\})$ *with* $\rho^\ell \cdot \mathcal{R}(\{r_1, r_2, z\})$ *in* Q *to produce the new precondition* $0.99 \leq \rho^\ell \cdot \mathcal{R}(\{r_1, r_2, z\})$.

Scalar Load/Store. The following equations present the rules for loads and stores from potentially approximate memory:

$$C_\psi(r_1 = \texttt{load}\ r_2, Q) = Q\,[\,\rho^{\chi(\eta(r_2))}_{ld} \cdot \mathcal{R}(\{\eta(r_2)\} \cup X)/\mathcal{R}(\{r_1\} \cup X)\,]$$
$$C_\psi(\texttt{store}\ r_1\ r_2, Q) = Q\,[\,\rho^{\chi(\eta(r_1))}_{st} \cdot \mathcal{R}(\{r_2\} \cup X)/\mathcal{R}(\{\eta(r_1)\} \cup X)\,]$$

These rules for load and store define the semantics of strong updates for scalar program variables. The rules use the auxiliary *register mapping* generated by the compiler

($\eta \in R \to V$) that maps the address operand register to the program variable that is read or written. The minimum reliability of a load from a potentially approximate variable, ρ_{ld}, is equal to the probability that the read from memory, the write to a cache location, and the read from that cache location all execute correctly, $\pi_1(\pi_{\text{mem}}(\psi)) \cdot \pi_1(\pi_{\$}(\psi)) \cdot \pi_2(\pi_{\$}(\psi))$. The reliability of a store to a potentially approximate variable, ρ_{st}, assuming a write-through cache, is equal to the reliability of a memory store, $\pi_2(\pi_{\text{mem}}(\psi))$.

***Example* 6** (Analysis of Scalar Store). *We analyze the statement* `store` r_1, r_2 *with the postcondition* $Q := 0.99 \leq \mathcal{R}(\{x\})$ *and the hardware configuration* ψ. *The statement stores the value of the register* r_2 *to the location in memory referred by the register* r_1. *We consider the case when* r_1 *holds the location of the variable* x, *i.e.,* $\eta(r_1) = x$.
First, the analysis computes the constant ρ *from* ψ. *Second, the analysis identifies that the register* r_1 *holds the address of* x *(using the map* η*) and finds the label* ℓ *that corresponds to the variable* x *(using the map* χ*). Third, the analysis generates the new reliability factor* $\mathcal{R}(\{r_2\})$ *by substituting* x *with* r_2. *Finally, it substitutes* $\mathcal{R}(\{x\})$ *with* $\rho^\ell \cdot \mathcal{R}(\{r_2\})$ *in* Q *to produce the precondition* $0.99 \leq \rho^\ell \cdot \mathcal{R}(\{r_2\})$.

Array Load/Store. The reliability constraint generation rule for stores to scalar variables provides a semantics for strong updates to memory. Updates to arrays, however, are weak in that a variable refers to multiple memory locations. The following reliability constraint generation rule defines the analysis for arrays:

$$C_\psi(r_{val} = \texttt{loada}\ r_{arr}\ r_{idx}, Q) = Q\,[\,\rho_{ld}^{\chi(\eta(r_{arr}))} \cdot \mathcal{R}(\{\eta(r_{arr}), r_{idx}\} \cup X) / \mathcal{R}(\{r_{val}\} \cup X)\,]$$

$$C_\psi(\texttt{storea}\ r_{arr}\ r_{idx}\ r_{val}, Q) = Q\,[\,\rho_{st}^{\chi(\eta(r_{arr}))} \cdot \mathcal{R}(\{r_{idx}, r_{val}\} \cup \{\eta(r_{arr})\} \cup X)\ /\ \mathcal{R}(\{\eta(r_{arr}))\} \cup X)\,]$$

The primary difference between this rule and that for strong updates is that the reliability of the array variable is included in the resulting reliability term (after substitution). Since the function $\eta(r_1)$ points to the same variable name for all elements of the array, this rule effectively treats updates to the potentially different array elements as an update to the single (scalar) variable.

***Example* 7** (Analysis of Array Store). *We analyze the same statement as in Example 6, but instead consider the case when* x *is an array variable.*
The analysis differs from the analysis for the scalar variables in the third step – the construction of the new reliability factor $\mathcal{R}(\{r_2, x\})$. *The generated precondition is therefore* $0.99 \leq \rho^\ell \cdot \mathcal{R}(\{r_2, x\})$, *which states that the update to the array* x *does not necessarily overwrite the previous updates (to a different or the same array element).*

Conditionals. The analysis of conditionals relies on the fact that the Rely base language has structured control flow and therefore the intermediate language keeps the conditional structure. The following reliability constraint generation rule implements the analysis for conditionals:

$$
\begin{aligned}
&C_\psi(\texttt{if } r_c \ inst' \ inst'', Q) \ = \\
&\quad \textbf{let } \{o_1, \ldots, o_k\} = \text{modified}(inst') \cup \text{modified}(inst') \\
&\quad \textbf{and } Q^* = Q\,[\mathcal{R}(\{r_c, o_1\} \cup X)/\mathcal{R}(\{o_1\} \cup X)] \ldots [\mathcal{R}(\{r_c, o_k\} \cup X)/\mathcal{R}(\{o_k\} \cup X) \textbf{ in} \\
&\qquad C_\psi(\ inst', Q^*\ ) \wedge C_\psi(\ inst'', Q^*\ )
\end{aligned}
$$

This rule uses the helper function modified $\in$ *Instr* $\rightarrow$ O, which denotes the set of operands (variables and registers) that are modified within a loop branch. To encode the probability that an incorrectly computed conditional may cause an incorrect value in each such operand, it adds the reliability of the conditional expression (r_c) to the joint reliability term containing the operand in the predicate Q. The final predicate is the conjunction of the predicates computed for each of the branches.

***Example* 8** (Analysis of Conditionals). *Consider* $\texttt{if } r_c \ \{r_3 = \texttt{add}^{\ell_1} \ r_1 \ 1\} \ \{r_3 = \texttt{sub}^{\ell_2} \ r_2 \ 1\}$ *and the postcondition* $Q := 0.9 \leq \mathcal{R}(\{r_3\})$. *The analysis first finds that* r_3 *is the only operand modified in the loop. Computing the predicate for the then branch yields* $Q_{then} := 0.9 \leq \rho_+^{\ell_1} \cdot \mathcal{R}(\{r_c, r_1\})$ *and for the else branch yields* $Q_{else} := 0.9 \leq \rho_-^{\ell_2} \cdot \mathcal{R}(\{r_c, r_2\})$. *The final predicate is then* $Q_{then} \wedge Q_{else}$.

***Example* 9** (Analysis of Conditionals). *Consider* $\texttt{if } r_c \ \{r_2 = \texttt{add}^{\ell_1} \ r_1 \ 1\} \ \{r_3 = \texttt{sub}^{\ell_2} \ r_1 \ 1\}$ *and the postcondition* $Q := 0.9 \leq \mathcal{R}(\{r_2, r_3\})$. *The analysis first finds that* r_2 *and* r_3 *are the operands modified in the loop. The predicate for the then branch is* $Q_{then} := 0.9 \leq \rho_+^{\ell_1} \cdot \mathcal{R}(\{r_c, r_1, r_3\})$ *and for the else branch is* $Q_{else} := 0.9 \leq \rho_-^{\ell_2} \cdot \mathcal{R}(\{r_c, r_1, r_2\})$. *The final predicate is then* $Q_{then} \wedge Q_{else}$.

Bounded Loops. Bounded loops are translated to the intermediate language as a sequence of conditional statements. Then, the analysis uses the rule for conditionals that we previously presented to compute the reliability. Chisel does not handle Rely programs with unbounded loops.

Final Precondition

For a given kernel, our analysis computes a precondition that is a conjunction of predicates of the form

$$\rho_{spec} \cdot \mathcal{R}(V_{spec}) \leq r(\ell_1, \ldots, \ell_n) \cdot \mathcal{R}(V),$$

where $\rho_{spec} \cdot \mathcal{R}(V_{spec})$ is a reliability factor for a developer-provided specification of an output and $r(\ell_1, \ldots, \ell_n) \cdot \mathcal{R}(V)$ is a lower bound on the output's reliability computed by the analysis, parameterized by the labels $\ell_1, \ldots, \ell_n$ of the candidate approximate operations.

Each ρ_{spec} is a real-valued constant and each r is, syntactically, a product of a real-valued constant and kinded reliabilities, i.e.,

$$r(\ell_1, \ldots, \ell_n) = \rho \cdot \Pi_k \, \rho_k^{\ell_k}. \tag{5.1}$$

The product operator iterates over the sequences of instructions that the analysis traversed. If this precondition is valid for a given kind configuration $\theta(\cdot)$, then that kind configuration satisfies the developer-provided reliability specification.

***Example* 10** (Analysis of a Function). *We consider a simple function written in the Rely language,* `int<0.99*R(x)> f(int x) { return x+3; }` *for which the compiler generates these assembly instructions:* `r0 = init ⟨x⟩; r1 = load r0; r2 = init 3; r3 = add`$^{\ell_+}$ `r1 r2; return r3;` *(the operator* $\langle$`x`$\rangle$ *denotes the stack offset of the variable* $\boldsymbol{x}$*).*
The kind configuration θ has two elements: ℓ_+ for the arithmetic operator and ℓ_x for the parameter x (which can be stored in exact or approximate memory). The analysis constructs the following preconditions for the instructions:

$$\begin{aligned}
Q_1 = C_\psi(\texttt{return } r_3,\ \mathit{True}) &:= 0.99 \cdot \mathcal{R}(\{x\}) \leq \mathcal{R}(\{r_3\}) \\
Q_2 = C_\psi(\texttt{add } r_1\ r_2,\ Q_1) &:= 0.99 \cdot \mathcal{R}(\{x\}) \leq \rho_{add}^{\ell_+} \cdot \mathcal{R}(\{r_1, r_2\}) \\
Q_3 = C_\psi(r_2 = \texttt{init } 3,\ Q_2) &:= 0.99 \cdot \mathcal{R}(\{x\}) \leq \rho_{add}^{\ell_+} \cdot \rho_{ld}^{\ell_x} \cdot \mathcal{R}(\{r_1\}) \\
Q_4 = C_\psi(r_1 = \texttt{load } r_0,\ Q_3) &:= 0.99 \cdot \mathcal{R}(\{x\}) \leq \rho_{add}^{\ell_+} \cdot \rho_{ld}^{\ell_x} \cdot \mathcal{R}(\{x\}) \\
Q_5 = C_\psi(r_0 = \texttt{init } \langle x \rangle,\ Q_4) &:= 0.99 \cdot \mathcal{R}(\{x\}) \leq \rho_{add}^{\ell_+} \cdot \rho_{ld}^{\ell_x} \cdot \mathcal{R}(\{x\})
\end{aligned}$$

Q_5 is the final precondition for the function. This derivation combines the rules for the instructions we previously described. We note that the analysis of load statement immediately inserts the variable x in the joint reliability factor (because of the mapping $\eta(r_0) = x$), and therefore the subsequent analysis of the instruction $r_0 =$ `init` $\langle x \rangle$ *does not modify the predicate.*

Soundness. The reliability analysis is sound with the respect to the paired execution semantics of the approximate computation. The soundness argument follows from the soundness of Rely [19]. It is also a special case of the soundness of the analysis we present in Section 5.3. When the kind configuration $\theta(\cdot)$ for the instructions is known, the analysis can substitute each $\theta(\ell)$ and check whether the final precondition is correct, using the same approach as in Rely.

Simplification

The number of constraints that the generator produces can, in principle, grow exponentially in the number of conditional statements in the program. However, in practice, the number of constraints can be significantly decreased by using a simplification of the constraints after each step of the algorithm [19]. Chisel extends Rely's simplification procedure, which uses the ordering property of the joint reliability factors and the subsumption property of the reliability predicates.

Ordering Of Joint Reliability Factors. Ordering enables comparing two joint reliability factors by comparing their sets of variables [19, Proposition 1]. Specifically, this proposition states that for the two sets of variables V and V_{spec},

$$V \subseteq V_{spec} \Rightarrow \mathcal{R}(V_{spec}) \leq \mathcal{R}(V). \tag{5.2}$$

Therefore, the reliability of any subset of a set of variables is greater than or equal to the reliability of the set as a whole. This property immediately extends from the sets of variables to the sets of operands (registers and variables).

Ordering of Labeled Reliabilities. Chisel operates on products of labeled reliabilities, which can also be ordered. Specifically, $\hat{\rho}_1^{\ell_1} \cdot \ldots \cdot \hat{\rho}_n^{\ell_n} \leq \rho_1^{\ell_1} \cdot \ldots \cdot \rho_m^{\ell_m}$ if $\{\ell_1, \ldots, \ell_m\} \subseteq \{\ell_1, \ldots, \ell_n\}$ and for each $\ell_i \in \{\ell_1, \ldots, \ell_n\}$, either $\hat{\rho}_i \leq \rho_i$ or ρ_i does not show up in the product on the right-hand side (in which case the reliability is by default equal to one).

Subsumption. This property defines the condition under which a predicate is trivially satisfied, given another more general predicate [19, Proposition 2]. Specifically, this proposition states that a predicate $\rho_1 \cdot \mathcal{R}(V_1) \leq \rho_2 \cdot \mathcal{R}(V_2)$ subsumes (i.e., soundly replaces) a predicate $r_1' \cdot \mathcal{R}(X_1') \leq \rho_2' \cdot \mathcal{R}(V_2')$ if $\rho_1' \cdot \mathcal{R}(V_1') \leq \rho_1 \cdot \mathcal{R}(V_1)$ and $\rho_2 \cdot \mathcal{R}(V_2) \leq \rho_2' \cdot \mathcal{R}(V_2')$. In Chisel, this proposition follows immediately from the ordering of joint reliability factors and the ordering of labeled reliabilities.

5.2.5 Optimization Constraint Construction

When the kind configuration $\theta(\cdot)$ is unknown, the final precondition that Chisel's generator produces represents a constraint that lists *all* approximation choices represented by θ. Then, each $\theta(\ell)$ is a variable that can be either 0 (reliable) or 1 (unreliable). The precondition parameterized by $\theta(\cdot)$ therefore represents all approximate versions of the program

that satisfy the developer's reliability specification. To generate the constraint for the optimization problem, Chisel analyzes separately the reliability degradation and joint reliability factors in each conjunct: 1) $\rho_{spec} \leq r(\ell_1, \ldots, \ell_n)$ and 2) $\mathcal{R}(V_{spec}) \leq \mathcal{R}(V)$.

Validity Checking. To check the validity of this precondition, we use the ordering property, from Equation 5.2. Therefore, Chisel can soundly ensure the validity of each inequality in the precondition by verifying that 1) $\rho_{spec} \leq r(\ell_1, \ldots, \ell_n)$ and 2) $V \subseteq V_{spec}$. Since V and V_{spec} are not parameterized by the labels ℓ, Chisel can immediately check if these set inclusion constraints are satisfied.

Constraint Construction. After checking the validity of the reliability factors, Chisel is left with the inequality

$$\rho_{spec} \leq r(\ell_1, \ldots, \ell_n). \tag{5.3}$$

The denotation of the reliability expression r is $\rho \cdot \Pi_k \ \rho_k^{\theta(\ell_k)}$. Recall that, given the kind configuration θ, the denotation of each ρ^ℓ from Equation 5.1 is $\rho^{\theta(\ell)}$.

Chisel therefore produces a final optimization constraint by taking the logarithm of both sides of Inequality 5.3:

$$\log(\rho_{spec}) - \log(\rho) \leq \sum_k \theta(\ell_k) \cdot \log(\rho_k). \tag{5.4}$$

The expression on the right side is linear with respect to all labels' kinds $\theta(\ell_k)$. The reliabilities ρ are constants and their logarithms can be immediately computed. Each label's kind is an (unknown) integer variable that can take a value 0 or 1.

5.3 Accuracy Constraint Construction

To exploit the capabilities of architectures that have variable-precision floating point units [40, 56, 120, 121], we now present Chisel's analysis that unifies reasoning about reliability and accuracy. Specifically, we extend reliability predicates with the ability to characterize the difference in a the values of variables between the kernel's exact and approximate executions. Then, our constraint generator produces linear expressions of kind configurations that characterize how the numerical error emerges and propagates through the kernel.

5.3.1 Accuracy Specification

Approximate Hardware Specification. For each approximate floating point operation op, we extend the definition of the hardware specification ψ from Section 4.2.1 to also include the accuracy specification of the variable-accuracy instructions. The specification of a variable-accuracy instruction consists of 1) the reliability r and 2) the number of mantissa bits that are computed fully accurately c, which determines the maximum absolute numerical error of the operation.

The approximate instructions have the following semantics. With probability at least r, an approximate arithmetic instruction produces a result that has a numerical error with bounded magnitude (determined by c). With probability at most $1 - r$, the operation can produce an arbitrarily inaccurate result.

Therefore, this specification combines the magnitude and the frequency of error. The source of the bounded error with small-magnitude is typically a simplified design of FPUs that requires less gates to build. The source of the unbounded errors occurring with small frequency is typically (like for the ALUs) the timing variation in the gates.

Function Specification. We extend the syntax of reliability specifications from the Rely base language to include a specification of acceptable accuracy loss. The extended specification has the following form:

```
float <d, r * R( d1 >= D(x1), ..., dn >= D(xn) )> f(...)
```

The constant `d` specifies the maximum acceptable difference between the results of the exact and approximate executions. The constant `r` specifies the probability with which the approximate execution will produce a result within distance `d` of the exact result. The constraints `di >= D(xi)` specify that the non-negative value `di` is the maximum absolute difference between the values of the function's parameter `xi` at the beginning of the exact and approximate kernel executions.

Interval Specification. We extend function specifications to enable developers to specify the intervals of values of a function's parameters. Because the accuracy analysis relies on an internal interval analysis, the precision of the results of this analysis depends on the precision of the intervals specified for the inputs to the function. To specify the parameter interval, a developer precedes a function's declaration with an annotation of the form `@interval(p,a,b)`, denoting that the parameter `p` is within the interval $[a, b]$.

5.3.2 Accuracy Predicates

We present the syntax and semantics of the predicates generated by the accuracy analysis:

$$\begin{array}{rcl} Q_A & := & d \geq A_e \mid Q_A \wedge Q_A \mid \text{True} \mid \text{False} \\ A_e & := & d \mid d \cdot \ell \mid d \cdot \Delta(o) \mid A_e + A_e \end{array}$$

An accuracy predicate Q_A is a conjunction of accuracy predicates or a comparison between a numerical constant d and an accuracy expression A_e, which has one of four forms:

- **Constant.** A term $d \in \mathbb{R}_0^+ \cup \{\infty\}$ represents an approximate operation's numerical error, which propagates to the computation's output. It is a non-negative number or a special constant ∞, which represents the maximum error. For the use in our analysis, we define that $[\![0 \cdot \infty]\!] = 0$ and for any $d > 0$, $[\![d \cdot \infty]\!] = \infty$.
- **Product of Constant and Label.** A term $d \cdot \ell$ represents a numerical error of an operation that can be either exact or approximate. The label ℓ encodes the choice between the exact and approximate version of an instruction: if $\theta(\ell) = 0$ (exact), then $d \cdot \ell$ is zero; if $\theta(\ell) = 1$ (approximate), then $d \cdot \ell$ is equal to d.
- **Product of Constant and Distance.** A term $d \cdot \Delta(o)$ represents a numerical error of an operand o (which is a register r or a variable v). Specifically, a *distance operator* $\Delta(o)$ relates the values of the operand o in an exact and an approximate execution.
- **Sum of Accuracy Expressions.** A term $A_{e1} + A_{e2}$ represents a sum of accuracy expressions A_{e1} and A_{e2}.

Figure 5.4 presents the denotational semantics of accuracy expressions and predicates. Accuracy expressions and predicates have a similar semantics to that of standard logical predicates over numerical expressions. The main point of departure is the semantics of the distance operator, which is the absolute difference between the value of a variable in an exact environment ε and its corresponding value in an approximate environment ε_a.

For notational purposes, we define implication as: $Q_{A1} \Rightarrow Q_{A2} \equiv [\![Q_{A1}]\!] \subseteq [\![Q_{A2}]\!]$.

5.3.3 Extended Reliability Predicates

To specify Chisel's extended reliability precondition generator, we embed accuracy predicates within the reliability predicates that we presented in Section 5.2.1. Specifically, we add a *generalized joint reliability factor*, $\mathcal{R}^*(Q_A)$. Figure 5.5 presents that this factor denotes the probability that the exact environment ε and an approximate environment ε_a sampled from φ together satisfy the accuracy predicate Q_A.

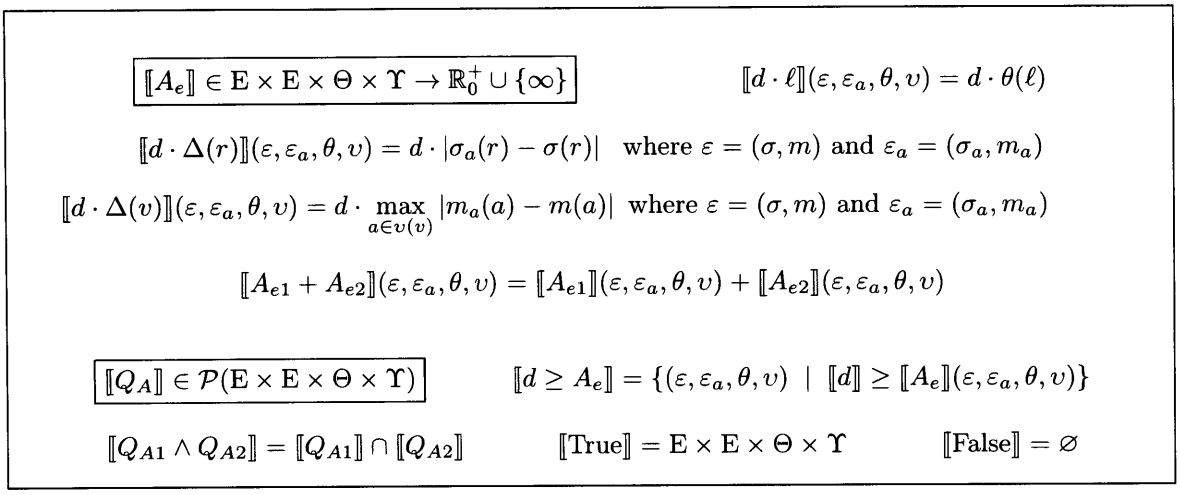


Figure 5.4: Semantics of Accuracy Predicates

$$[\![\mathcal{R}^*(Q_A)]\!](\varepsilon, \varphi, \theta, \upsilon) = \sum_{\varepsilon_a \in \mathcal{E}(\varepsilon, \theta, \upsilon, Q_A)} \varphi(\varepsilon_a),$$

$$\text{where } \mathcal{E}(\varepsilon, \theta, \upsilon, Q_A) = \{\, \varepsilon_a \mid (\varepsilon, \varepsilon_a, \theta, \upsilon) \in [\![Q_A]\!] \,\}.$$

Figure 5.5: Generalized Joint Reliability Factor

The syntax of the extended reliability factors (R_f) and predicates (P') extends the definitions of reliability factors (R_f) and predicates from Section 5.2.1:

$$R'_f := R_f \mid \mathcal{R}^*(Q_A)$$
$$P' := R'_f \leq R'_f \mid P' \wedge P'.$$

This definition of joint reliability factors subsumes the definition of the standard joint reliability factors $\mathcal{R}(O)$ (Section 5.2.1, Definition 3). Specifically, the set of variables that have the same value in the exact and approximate program executions can be represented using accuracy predicates that bound the absolute difference of each variable by zero:

$$[\![\mathcal{R}(O)]\!](\varepsilon, \varphi, \theta, \upsilon) = [\![\mathcal{R}^*(\bigwedge_{o \in O} 0 \geq \Delta(o))]\!](\varepsilon, \varphi, \theta, \upsilon).$$

Alternatively, we can also represent the joint reliability factor $\mathcal{R}(O)$ with the accuracy predicate that bounds the sum of the absolute differences of all variables by zero:

$$[\![\mathcal{R}(O)]\!](\varepsilon, \varphi, \theta, \upsilon) = [\![\mathcal{R}^*(0 \geq \sum_{o \in O} \Delta(o))]\!](\varepsilon, \varphi, \theta, \upsilon).$$

5.3.4 Extended Reliability Precondition Generator

We now present the precondition generator of extended reliability predicates at the level of the statements in the intermediate language. This analysis is applicable to kernel computations with conditionals and bounded loops. The combined accuracy and reliability precondition generator takes as input a statement s and an extended reliability postcondition P and generates a precondition P', such that if P' holds before the paired execution of the statement s, then P holds after the paired execution of s. Below we present the three main components of the precondition generator.

Interval Analysis. Before running the precondition generator, the analysis runs an auxiliary interval analysis (Section 5.3.5). The interval analysis computes the intervals of instruction results under the assumption that all instructions in the kernel have reduced precision. This way, this analysis can conservatively capture how the error propagates through the computation, even if the subsequent analysis selects that some of the instructions should eventually be exact.

Analysis of Arithmetic Expressions. For each arithmetic operator, the analysis computes an accuracy predicate that quantifies 1) the error that *emerges* as a result of imprecise instruction and 2) the error that *propagates* from the instruction's operands, which were computed by the previous approximate computation (Section 5.3.6).

The accuracy predicate separates the error in the emerged error and propagated error, which enables the analysis to quantify the effect of each potentially approximate operation on the kernel's output. The emerged error term is a function of the operation's label ℓ and the maximum sensitivity of the output to the error introduced by instruction approximation.

Sensitivity is a multiplicative constant, consisting of the maximum error of the operation and the terms that are upper bound on how subsequent instructions amplify this error. In principle, sensitivity is affected by the result of other potentially approximate operations, and as such would be a function of these instructions' labels, $\ell_1, \ldots, \ell_k$. However, multiplication of labels would make the optimization problem non-linear. Therefore, the accuracy analysis computes the maximum sensitivity, which is a constant computed under the assumption that all operations are approximate (i.e., $\theta(\ell_1) = 1, \ldots, \theta(\ell_k) = 1$). This constant can be computed directly from the auxiliary interval analysis (which operates under the same assumption that all operations are approximate).

By computing this sensitivity constant, the analysis *linearizes* the accuracy predicate. By assigning the labels to the errors emerging from specific instructions, the accuracy predicates

can identify those instructions that (in the worst-case) produce unacceptable absolute error. The solver can then select that such operations should execute exactly.

Reliability and Accuracy Analysis. In the third step, the precondition generator combines the accuracy predicate with a reliability degradation expression (which we presented in Section 5.2). Specifically, Section 5.3.7 presents how the analysis uses the generalized joint reliability factor to embed the accuracy predicate to specify a more relaxed model of reliability. In this reliability model, any computation that satisfies the accuracy predicate is considered reliable, even if it is not exact.

5.3.5 Auxiliary Interval Analysis

To compute the absolute error induced by variable-accuracy arithmetic operations (given the number of accurately computed mantissa bits), an accuracy analysis requires the intervals of the operation's inputs. Therefore, we define an auxiliary interval analysis that computes the intervals of arithmetic instructions computed within the kernel. These intervals include the maximum absolute errors induced by the variable-accuracy floating point operations.

The analysis produces a mapping $\mathcal{I} : \mathcal{L} \rightarrow (\text{Float} \times \text{Float}) + \textit{Unbounded}$, which yields the interval of values to which each instruction operand (identified by its label $\ell \in \mathcal{L}$) evaluates. The set Float contains all floating point numbers that the target platform can represent. A special symbol *Unbounded* indicates that the interval is unbounded (due to e.g., a possible overflow or divide by zero).

The analysis operates in a forward fashion, using the standard rules of interval arithmetic. To provide a conservative estimate of the error that the approximate execution may produce, for every arithmetic operation $r = op^{\ell}\ r_1\ r_2$ the interval analysis extends the computed interval of the result of the exact operation $[a, b]$ with an error term δ (which depends on the operation), which represents the maximum absolute error of the approximate operation. The resulting interval is therefore $\mathcal{I}(\ell) = [a - \delta, b + \delta]$. The computation of conservative intervals is inspired by the analysis presented in [31].

To compute the error term for an arithmetic operation, the analysis uses the function $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(r_1, r_2)$, which returns the maximum error of the operation op when it operates on only a fraction of the inputs' mantissa bits and the intervals of the operands are $\mathcal{I}(\mathsf{loc}(r_1))$ and $\mathcal{I}(\mathsf{loc}(r_2))$. The function loc returns the label corresponding to the register use. If any operand interval is unbounded, then the result interval is also unbounded. The analysis computes the intervals for computations in which the approximate execution does not cause control-flow divergence from the original execution.

We compute the maximum error as follows. If a floating point instruction computes a result with c exact mantissa bits, then machine epsilon (the distance between the value 1.0 and the next adjacent value) is $2^{-(c-1)}$. If the interval of the output computed for intervals $\mathcal{I}(\mathsf{loc}(r_1))$ and $\mathcal{I}(\mathsf{loc}(r_2))$ is $[a_r, b_r]$, and $\max(|a_r|, |b_r|) \leq 2^w$ (where w is the minimal such exponent), then the maximum error is lower than or equal to $0.5 \cdot 2^{w-(c-1)}$, where the constant 0.5 comes from the rounding-to-the-nearest semantics of floating point instructions.

5.3.6 Analysis of Arithmetic Instructions

The function $AE_{\psi,\mathcal{I}} \in S \to A_e$ produces an expression that bounds the absolute error of an arithmetic instruction:

$$\begin{aligned} AE_{\psi,\mathcal{I}}(r = op^{\ell}\ r_1\ r_2) \quad = \quad & \pi_1(\mathsf{propagation}_{op,\mathcal{I}}(r_1, r_2)) \cdot \Delta(r_1) + \\ & \pi_2(\mathsf{propagation}_{op,\mathcal{I}}(r_1, r_2)) \cdot \Delta(r_2) + \\ & \ell \cdot \mathsf{maxerr}_{op,\psi,\mathcal{I}}(r_1, r_2) \end{aligned}$$

Error Propagation. The function $\mathsf{propagation}_{op,\mathcal{I}}(r_1, r_2)$ returns a pair of real-valued *error propagation coefficients* (k_1, k_2) that specify how sensitive the result of the operation is to the changes of the first and the second operand, respectively.

To compute the coefficients for each operation, we use the observation that for a differentiable function $f(x, y)$ defined on a bounded interval and inputs with errors $\hat{x} = x + \delta_x$ and $\hat{y} = y + \delta_y$, one can show that $|f(x, y) - f(\hat{x}, \hat{y})| \leq k_1 \cdot |\delta_x| + k_2 \cdot |\delta_y|$. The constants $k_1 = \max_{x,y} \left|\frac{\partial f(x,y)}{\partial x}\right|$ and $k_2 = \max_{x,y} \left|\frac{\partial f(x,y)}{\partial y}\right|$ can be computed from the input intervals when the partial derivatives of f are bounded. Note that the input intervals include the bounds for the errors δ_x and δ_y.

We can use this observation to specify the error propagation functions for the four arithmetic operations:

$$\begin{aligned} \mathsf{propagation}_{+,\mathcal{I}}(r_1, r_2) &= (1, 1) \\ \mathsf{propagation}_{-,\mathcal{I}}(r_1, r_2) &= (1, 1) \\ \mathsf{propagation}_{*,\mathcal{I}}(r_1, r_2) &= (\max_{y \in \mathcal{I}_{r_2}} |y|, \max_{x \in \mathcal{I}_{r_1}} |x|\ ) \\ \mathsf{propagation}_{\div,\mathcal{I}}(r_1, r_2) &= (\max_{y \in \mathcal{I}_{r_2}} |1/y|, \max_{x \in \mathcal{I}_{r_1}, y \in \mathcal{I}_{r_2}} |x/y^2|\ ) \text{ when } 0 \notin \mathcal{I}_{r_2}. \end{aligned}$$

Recall that the conservative interval analysis incorporates the maximum error that can propagate from the operands. Therefore, the intervals of the operands $\mathcal{I}_{r_1} = \mathcal{I}(\mathsf{loc}(r_1))$ and $\mathcal{I}_{r_2} = \mathcal{I}(\mathsf{loc}(r_2))$ incorporate the upper bounds for the errors in the operands. If either

interval is unbounded or the divisor's interval includes 0, then the corresponding coefficient will be infinity (∞), indicating that the operand's value is *critical*, i.e., the kernel's result is highly sensitive to its change.

***Example* 11** (Error of Multiplication). *We analyze the instruction* $r = \texttt{mul}\ r_1\ r_2$, *given the intervals* $\mathcal{I}_{r_1} = \mathcal{I}(\mathsf{loc}(r_1)) = [-2, 1]$ *and* $\mathcal{I}_{r_2} = \mathcal{I}(\mathsf{loc}(r_2)) = [0, 8]$ *and the exact operator* `mul`.
The function propagation *returns the pair* $(8, 2)$, *meaning that the error that propagates through* r_1 *can be amplified by a factor of 8 (maximum absolute value of* r_2*) and the error that propagates through* r_2 *can be amplified by a factor of 2 (maximum absolute value of* r_1*). The analysis therefore produces the accuracy expression* $8 \cdot \Delta(r_1) + 2 \cdot \Delta(r_2)$.

Error Induced by Approximation. The analysis uses the function $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(r_1, r_2)$ to compute the maximum error induced by the approximate arithmetic instruction when the inputs are in $\mathcal{I}(\mathsf{loc}(r_1))$ and $\mathcal{I}(\mathsf{loc}(r_2))$. If either of the intervals is unbounded, then the function returns ∞.

The propagation and approximation-induced errors are additive because for two continuous functions f and $\hat{f}$, we have $|f(x, y) - \hat{f}(\hat{x}, \hat{y})| \leq |f(x, y) - f(\hat{x}, \hat{y})| + |f(\hat{x}, \hat{y}) - \hat{f}(\hat{x}, \hat{y})|$ from the triangle inequality. Therefore, the total absolute error is bounded by the sum of the error propagating through the operands, characterized by the propagation coefficients from $\mathsf{propagation}_{op,\mathcal{I}}(\cdot)$, and the induced error, $\mathsf{maxerr}_{op,\psi,\mathcal{I}}(\cdot)$. To control whether to approximate the operation, the generator multiplies the induced error with the operation's label.

***Example* 12** (Error of Multiplication). *We analyze the expression* $r = \texttt{mul}\ r_1\ r_2$ *introduced in Example 11. This operator computes the result with* $c = 8$ *exact mantissa bits.*
The interval of the result of the multiplication is $[-16, 16]$. *The maximum absolute error on this interval is* 0.0625. *Therefore, the accuracy generator produces an expression* $0.0625 + 8 \cdot \Delta(a) + 2 \cdot \Delta(b)$.

5.3.7 Generalized Reliability and Accuracy Analysis

This section presents the rules for the precondition generator for intermediate-language statements, $C^*_{\psi,\mathcal{I}} \in S \times P' \to P'$. The precondition generator for statements operates backwards, from the end to the beginning of the kernel function. The analysis rules for instructions are analogous to those from the reliability analysis in Section 5.2. The main difference between the two analyses is in the propagation of the accuracy predicate Q_A within the generalized reliability factors, as opposed to propagating sets of variables.

Initial Postcondition

Just like the reliability generator (Section 5.2), it transforms the extended reliability predicate Q_R, starting from the predicate that is the conjunction of the terms

$$Q_0 := \bigwedge_i \rho_{spec,i} \cdot \mathcal{R}^*(Q_{spec,i}) \leq \mathcal{R}^*(d_{spec,i} \geq \Delta(v_i))$$

for each kernel's array parameter v_i or $Q_0 :=$ True if there are none. The analysis begins from the return statement.

Return. The following equation presents the rule for the return instruction:

$$C^*_{\psi,\mathcal{I}}(\texttt{return } r_{ret},\, Q_0) = \textbf{let } Q_A = d_{spec} \geq \Delta(r_{ret}) \textbf{ in } \rho_{spec} \cdot \mathcal{R}(V_{spec}) \leq \mathcal{R}^*(Q_A) \wedge Q_0$$

The analysis of the return statement first generates a new accuracy predicate Q_A, which states that the absolute difference between the values of r_{ret} in the exact and approximate executions should be less than the accuracy specification $\Gamma_A(\text{ret}) = d_{spec}$. The analysis then generates a reliability predicate by relating the kernel's acceptable reliability $\Gamma_R(\text{ret}) = (\rho_{spec}, V_{spec})$ with the probability that the predicate Q_A is correct.

***Example* 13.** *We analyze the function with specification* `int <0.1, 0.99*R(0.2>=D(x))>` *and the return instruction* `return` r_1.
For the return statement, the analysis produces $0.99 \cdot \mathcal{R}^*(0.2 \geq \Delta(r_1)) \leq \mathcal{R}^*(0.1 \geq \Delta(r_1))$. *The left-hand side comes from the specification. The right-hand side is generated for the return statement's expression.*

Precondition Generator for Instructions

We use the *conditional substitution* rule of the following form:

$$\textbf{let } Q'_A = Q_A\,[\,A\,/\,B\,] \textbf{ incond } Q_R\,[\,\rho \cdot \mathcal{R}^*(Q'_A)\,/\,\mathcal{R}^*(Q_A)\,].$$

If the accuracy predicate Q_A contains a sequence of symbols B, the first substitution replaces B with a sequence A to produce Q'_A,. The rule then, since $Q'_A \neq Q_A$, performs the second substitution to replace $\mathcal{R}^*(Q_A)$ with $\mathcal{R}^*(Q'_A)$, optionally multiplied by a constant factor ρ. If, on the other hand Q_A does not contain B, then $Q'_A = Q_A$, and the rule does not apply the second substitution (in Q_R).

Initialization and Sequence. The following equations present the rules for initializing a register with a constant and a sequence of instructions:

$$C^*_{\psi,\mathcal{I}}(r = \texttt{init}\ \ n,\ Q) = \textbf{let}\ Q'_A = Q_A[0/\Delta r]\ \textbf{incond}\ Q\,[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)\,]$$
$$C^*_{\psi,\mathcal{I}}(\texttt{inst1; inst2},\ Q) = C^*_{\psi,\mathcal{I}}(\,\texttt{inst1},\ C^*_{\psi,\mathcal{I}}(\texttt{inst2},\ Q)\,)$$

The initialization rule removes the occurrence of the register r in the accuracy predicate because the constant is always passed to a register exactly. The sequence rule operates backward – it first analyzes the second instruction, and passes its computed precondition as the postcondition of the previous instruction.

ALU/FPU. The following equation presents the generator rule for ALU/FPU operations:

$$C^*_{\psi,\mathcal{I}}(r = op^{\ell}\ \ r_1\ \ r_2,\ Q)\ = \textbf{let}\ Q'_A = Q_A[AE_{\psi,\mathcal{I}}(r = op^{\ell}\ r_1 r_2)\,/\,\Delta(r)]\ \textbf{incond}$$
$$Q\,[\rho^{\ell}_{op}\cdot\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)\,]$$

This rule substitutes the expression for the accuracy of the arithmetic operation, which is a function of the distances of the input registers and the label ℓ, instead of the distance of the output register r. In addition to the changes in the accuracy predicate Q_A, the rule updates the generalized reliability predicate with the factor ρ^{ℓ}_{op}, which denotes the probability that the operation executed correctly (same as in the reliability analysis in Section 5.2.4).

***Example* 14.** *We analyze the instruction* `r = mul`$^{\ell}$ r_1 r_2 *and the corresponding postcondition* $Q := 0.99 \leq \mathcal{R}^*(0.5 \geq 2\cdot\Delta(r))$. *The intervals of* r_1 *and* r_2 *are the same as in Example 11.*
First, the accuracy predicate generator calls the function $AE_{\psi,\mathcal{I}}$ *which produces the expression* $0.0625\cdot\ell + 8\cdot\Delta(r_1) + 2\cdot\Delta(r_2)$ *(according to Example 12). To produce the accuracy predicate, the analysis substitutes the term* $\Delta(r)$ *with the generated expression in the accuracy predicate* $0.5 \geq 2\cdot\Delta(r)$. *The generated accuracy predicate is therefore* $0.5 \geq 0.125\cdot\ell + 16\cdot\Delta(r_1) + 4\cdot\Delta(r_2)$. *The multiplication* $0.125\cdot\ell$ *encodes the choice of approximating the operator* `mul`$^{\ell}$.
To produce the new combined predicate, the analysis computes the reliability expression for the instruction that stores the result in r. *This reliability is* ρ^{ℓ}_{mul}. *Finally, the analysis performs the substitution to construct the final precondition* $0.99 \leq \rho^{\ell}_{mul}\cdot\mathcal{R}^*(0.5 \geq 0.125\cdot\ell + 16\cdot\Delta(r_1) + 4\cdot\Delta(r_2))$.

Scalar Load/Store. The following equations present the rules for loads and stores from potentially approximate scalars:

$$C^*_{\psi,\mathcal{I}}(r_{val} = \texttt{load}\ r_{addr},\ Q) = \textbf{let}\ Q'_A = Q_A[\Delta(\eta(r_{addr}))\ /\ r_{val}] \wedge 0 \geq \Delta(r_{addr})\ \textbf{incond}$$
$$Q\,[\,\rho_{ld}^{\chi(\eta(r_{addr}))} \cdot \mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)\,]$$
$$C^*_{\psi,\mathcal{I}}(\texttt{store}\ r_{addr}\ r_{val},\ Q) = \textbf{let}\ Q'_A = Q_A[\Delta(\eta(r_{addr}))\ /\ \Delta(r_{val})] \wedge 0 \geq \Delta(r_{addr})\ \textbf{incond}$$
$$Q\,[\,\rho_{st}^{\chi(\eta(r_{addr}))} \cdot \mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)\,]$$

This rule is analogous to the rule in the reliability analysis. They replace the element referring to the memory operation's destination location with the source location. Specifically, the load instruction replaces the distance of the register r_{val} with the variable $\eta(r_{addr})$ referred to by the address stored in the register r_{addr}. The store instruction replaces the distance of the variable $\eta(r_{addr})$ with the distance of the register r_{val}.

Array Load/Store. The following equations present the rules for loads and stores from potentially approximate arrays:

$$C^*_{\psi,\mathcal{I}}(r_{val} = \texttt{loada}\ r_{arr}\ r_{idx},\ Q) =$$
$$\textbf{let}\ Q'_A = Q_A[\Delta(\eta(r_{arr}))\ /\ \Delta(r_{val})\,] \wedge 0 \geq \Delta(r_{arr}) + \Delta(r_{idx})\ \textbf{incond}$$
$$Q\,[\,\rho_{ld}^{\chi(\eta(r_{arr}))} \cdot \mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)\,]$$
$$C^*_{\psi,\mathcal{I}}(\texttt{storea}\ r_{arr}\ r_{idx}\ r_{val},\ Q) =$$
$$\textbf{let}\ Q'_A = Q_A[\Delta(r_{val}) + \Delta(\eta(r_{arr}))\ /\ \Delta(\eta(r_{arr}))] \wedge 0 \geq \Delta(r_{arr}) + \Delta(r_{idx})\ \textbf{incond}$$
$$Q\,[\,\rho_{st}^{\chi(\eta(r_{arr}))} \cdot \mathcal{R}^*(Q'_A)\ /\ \mathcal{R}^*(Q_A)\,]$$

The rule for load ensures that both the address and the index are computed numerically precisely, by bounding the sum of non-negative $\Delta(r_{arr})$ and $\Delta(r_{idx})$ by zero. The rule for store, in addition, performs a weak update by replacing $\Delta(\eta(r_{arr}))$ with $\Delta(r_{val})+\Delta(\eta(r_{arr}))$.

Conditional. The following equation presents the rule for the conditional intermediate statement:

$$
\begin{array}{l}
C^*_{\psi,\mathcal{I}}(\texttt{if } r_c \; inst' \; inst'', Q) \; = \\
\quad \textbf{let } \{o_1, \ldots, o_k\} = \text{modified}(inst') \cup \text{modified}(inst') \\
\quad \textbf{and if } \text{contains\_one}(Q_A, \{\Delta(o_1), \ldots, \Delta(o_k)\}) \textbf{ then} \\
\quad\quad \textbf{let } Q'_A = Q_A \wedge 0 \geq \Delta(r_c) \textbf{ incond} \\
\quad\quad\quad C^*_{\psi,\mathcal{I}}(\, inst', Q[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)]\,) \wedge C^*_{\psi,\mathcal{I}}(\, inst'', Q[\mathcal{R}^*(Q'_A)/\mathcal{R}^*(Q_A)]\,)
\end{array}
$$

This rule appends the accuracy predicate $0 \geq \Delta(r_c)$ to every accuracy predicate that references at least one of the operand distance $\Delta(o_1), \ldots, \Delta(o_k)$. This additional predicates ensures that the instructions that compute the condition are not approximated and cause an approximate execution to follow a different branch of the execution.

***Example* 15** (Analysis of Conditionals). *Consider* `if` (r_c) $r_1 =$ `init 1; else` $r_2 =$ `init 3;` *with the postcondition* $Q := 0.99 \leq \mathcal{R}^*(\,0.1 \geq \Delta(r_1) + \Delta(r_2)\,)$.

The analysis considers each of the branches independently. The analysis of the then-branch 1) ensures that the approximation in r_c *does not change the execution path and 2) substitutes* $\Delta(r_1)$ *with 0. Therefore,* $Q^{pre}_{then} := 0.99 \leq \mathcal{R}^*(\,0.1 \geq \Delta(r_2) \wedge 0 \geq \Delta(r_c)\,)$. *Analogously, the analysis of the else-branch yields* $Q^{pre}_{else} := 0.99 \leq \mathcal{R}^*(\,0.1 \geq \Delta(r_1) \wedge 0 \geq \Delta(r_c)\,)$. *The condition* $0 \geq \Delta(r_c)$, *then it would be possible for the approximate execution to switch between the then and else branches (when the distance for each changed register would be 2.0 instead of 0.0). Finally,* $Q_{pre} := Q^{pre}_{then} \wedge Q^{pre}_{else}$.

Final Precondition

The generator produces a precondition that is a conjunction of terms of the form:

$$\rho_{spec} \cdot \mathcal{R}^*(Q_{spec}) \leq r(\ell_1, \ldots, \ell_n) \cdot \mathcal{R}^*(Q_A(\ell_1, \ldots, \ell_n)).$$

The predicate Q_{spec} (given by the specification) is a conjunction of terms of the form

$$d \geq \Delta(v), \tag{5.5}$$

where each d is a constant and each v is a function parameter.

The accuracy predicate Q_A (produced by the precondition generator) is a conjunction of terms of the form

$$d_{spec} \geq \sum_j \Delta(v_j) \cdot \prod_l d_{j,l} + \sum_k \ell_k \cdot \prod_l d_{k,l}\,. \tag{5.6}$$

The constant d_{spec} comes from the specification and the analysis computes coefficients $d_{j,l}$ and $d_{k,l}$. The first sum on the right side of the inequality represents how the error in the parameters propagates to the output and the second sum represents the error caused by the approximate execution of the arithmetic operators.

5.3.8 Optimization Constraint Construction

If the final precondition generated by the analysis is valid, then the program satisfies its accuracy specification. The validity problem for a precondition leads to a natural method for generating an optimization constraint that limits the set of possible kind configurations of the program to only those that satisfy the program's accuracy specification.

Predicate Validity. Similar to the procedure in Section 5.2.5, we demonstrate the validity of each of the final precondition's conjuncts,

$$\rho_{spec} \cdot \mathcal{R}^*(Q_{spec}) \leq r(\ell_1, \ldots, \ell_n) \cdot \mathcal{R}^*(Q_A^{post}(\ell_1, \ldots, \ell_n)),$$

by showing that 1) the reliability coefficient on the right side of the inequality is bounded from below by that on the left side, specifically that $\rho_{spec} \leq r(\ell_1, \ldots, \ell_n)$, and 2) the generalized joint reliability factor on the left side of the inequality is bounded above by that on the right side, specifically that $\mathcal{R}^*(Q_{spec}) \leq \mathcal{R}^*(Q_A)(\ell_1, \ldots, \ell_n)$.

Bounding the reliability coefficients (and generating appropriate optimization constraints) is the same as in Section 5.2.5. To bound the generalized reliability factors, we generalize the ordering property for joint reliability factors (Equation 5.2) as follows:

Proposition 1 (Generalized Reliability Factor Ordering).

$$\textit{If } Q_{A1} \Rightarrow Q_{A2} \textit{ then } \mathcal{R}^*(Q_{A1}) \leq \mathcal{R}^*(Q_{A2}).$$

This property follows from the fact that, if Q_{A1} implies Q_{A2}, then the set of approximate program environments that satisfy the predicate Q_{A1} is a subset of the environments that satisfy the predicate Q_{A2}. Therefore, $\mathcal{R}^*(Q_{A1})$, the probability of the environments satisfying Q_{A1}, must be less than or equal to $\mathcal{R}^*(Q_{A2})$, the probability of the environments satisfying Q_{A2}. Given this ordering property, the analysis can also use the same simplification procedure from Section 5.2.4.

Constraint Construction. Given the generalized reliability factor ordering, Chisel's goal is to generate an optimization constraint that ensures that $Q_{spec} \Rightarrow Q_A$ (which therefore

ensures that the corresponding conjunct in the precondition is valid). Chisel constructs this constraint via the observation that Q_{spec} has the form $\bigwedge_j d_j \geq \Delta(v_j)$ (Section 5.3.7, Equation 5.5). Therefore, it is sound to replace each occurrence of $\Delta(v_j)$ in Q_A with the corresponding d_j, yielding a predicate of the form:

$$d_{spec} \geq \sum_j d_j \cdot \prod_l d_{j,l} + \sum_k \ell_k \cdot \prod_l d_{k,l}. \tag{5.7}$$

The constraint generator takes this accuracy predicate and constructs the optimization constraint. First, it rearranges terms and simplifies numerical constants ($d^* = \sum_j d_j \cdot \prod_l d_{j,l}$ and $d_k^* = \prod_l d_{k,l}$). Since $d_k^* \cdot \ell_k$ denotes the multiplication of the constant d_k^* and the kind configuration $\theta(\ell_k)$, for each conjunct, the generator produces the constraint:

$$d_{spec} - d^* \geq \sum_k d_k^* \cdot \theta(\ell_k). \tag{5.8}$$

Identifying Critical Operations. As it generates the optimization constraint, the constraint generator identifies all accuracy expressions in which the coefficient d_k^* has the value ∞ (Section 5.3.6). Such expressions indicate that small deviations in the result of an intermediate operation or a variable value may cause a large deviation of the kernel's output. The constraint generator sets the corresponding kind configuration $\theta(\ell_k)$ to 0 (exact) and removes all terms with such assigned configurations from the final accuracy constraints.

5.3.9 Soundness

Chisel's extended reliability precondition generator is sound with respect to the paired execution semantics. This generator contains both reliability and accuracy predicates, related through the generalized joint reliability factor $\mathcal{R}^*(Q_A^{post})$. Below, we present the main steps of the proof for the soundness of the generator.

Theorem 1 (Soundness). *If postcondition P_{post} is an extended reliability predicate and the precondition $P_{pre} = C^*_{\psi,\mathcal{I}}(s, P_{post})$, then for all inputs that belong to intervals in $\mathcal{I}$,*

$$\psi, \theta, \chi, \upsilon, \mathcal{I} \models \{P_{pre}\}\ s\ \{P_{post}\}$$

The theorem states that if the pair $\langle \varepsilon, \varphi \rangle$ satisfies the precondition generated by the analysis, then the paired execution of the statement s produces the final pair $\langle \varepsilon', \varphi' \rangle$, which satisfies the postcondition.

As a reminder, a paired execution starts from the pair $\langle \varepsilon, \varphi \rangle$, where ε is the initial environment of the exact execution and φ is a distribution of approximate environments. Every environment ε_a such that $\varphi(\varepsilon_a) > 0$ is the initial environment of the approximate execution. The paired execution ends in the pair $\langle \varepsilon', \varphi' \rangle$, where ε' is the final environment of the exact execution and φ' is the distributionn of the final approximate environments. Every ε'_a such that $\varphi'(\varepsilon'_a) > 0$ is the reachable final environment of an approximate execution.

Previously, Carbin showed that the Rely precondition generator, that contains ordinary joint reliability factors of the form $\mathcal{R}(X)$ is sound [15]. The proof in this section shows the soundness of the predicates with generalized joint reliability factors $\mathcal{R}^*(Q_A)$ containing an accuracy predicate Q_A. To present the reasoning about the extended reliability analysis, we focus on the analysis of arithmetic instructions.

Lemma 2 (Soundness of Accuracy Expression). *Let the registers r_1 and r_2 have the values in the ranges specified by the interval analysis $\mathcal{I}$ and let $AE_{\psi,\mathcal{I}}(r = \mathsf{add}^{\ell}\ r_1\ r_2) = \Delta(r_1) + \Delta(r_2) + \ell \cdot \mathsf{maxerr}(r_1, r_2)$. Then, after executing the instruction, $AE_{\psi,\mathcal{I}}(r = \mathsf{add}^{\ell}\ r_1\ r_2) \geq \Delta(r)$.*

Proof. After the execution of the instruction, for all ε'_a that satisfy input intervals, specifically, $[\![\Delta(r_1)]\!](\varepsilon', \varepsilon'_a, \theta, \upsilon) = |\pi_1(\varepsilon')(r_1) - \pi_1(\varepsilon'_a)(r_1)|$ and $[\![\Delta(r_2)]\!](\varepsilon', \varepsilon'_a, \theta, \upsilon) = |\pi_1(\varepsilon')(r_2) - \pi_1(\varepsilon'_a)(r_2)|$. Likewise, $[\![\Delta(r)]\!](\varepsilon', \varepsilon'_a, \theta, \upsilon) = |\pi_1(\varepsilon')(r) - \pi_1(\varepsilon'_a)(r)|$. First, let us consider the case when the operator add^{ℓ} is exact. Then,

$$\begin{aligned} |\pi_2(\varepsilon')(r) - \pi_2(\varepsilon'_a)(r)| &= |\pi_1(\varepsilon')(r_1) + \pi_1(\varepsilon')(r_2) - \big(\pi_1(\varepsilon'_a)(r_1) + \pi_1(\varepsilon'_a)(r_2)\big)\,| \\ &\leq |\pi_1(\varepsilon')(r_1) - \pi_1(\varepsilon'_a)(r_1)| + |\pi_1(\varepsilon')(r_2) - \pi_1(\varepsilon'_a)(r_2)|. \end{aligned}$$

and, therefore, it follows that $\Delta(r) \leq \Delta(r_1) + \Delta(r_2)$.

We now consider the error induced by the approximate execution of the operator add^{ℓ}. In this case, $\Delta(r) \leq \Delta(r_1) + \Delta(r_2) + m$, where the absolute error bound is $m = \mathsf{maxerr}(r_1, r_2)$. The error of the inputs on the input intervals (that include possible noise coming from the inputs r_1 and r_2) is bounded by m, whose construction we discussed in Section 5.3.5.

Finally, the analysis returns the term $\ell \cdot m$ to capture the error induced in both the exact and approximate executions. If the instruction executes exactly (i.e., $\theta(\ell) = 0$), then $[\![\ell \cdot m]\!] = 0$. Similarly, if the instruction executes approximately (i.e., $\theta(\ell) = 1$), then $[\![\ell \cdot m]\!] = m$. The analysis for the other arithmetic operators proceeds similarly, using the discussion from Section 5.3.6. □

Lemma 3 (Soundness of Addition with Accuracy Predicates). *If the analysis computes the precondition* $Q_A^{pre} = Q_A^{post}[AE_{\psi,\mathcal{I}}(r \texttt{ = add } r_1\ r_2)/\Delta(r)]$ *then*

$$\begin{aligned}\psi, \theta, \chi, \upsilon, \mathcal{I} \ \models\ & \{A \leq \rho_{add}^{\ell} \cdot \mathcal{R}^*(Q_A^{pre})\} \\ & r \texttt{ = add } r_1\ r_2\texttt{;} \\ & \{A \leq \mathcal{R}^*(Q_A^{post})\}\end{aligned}$$

To express the proof, we first define several auxiliary sets, using the same names as in [15]:

- **Approximate input environments:** For the starting environment ε, $ins(\varepsilon) = \{\varepsilon_a \mid (\varepsilon, \varepsilon_a, \theta, \upsilon) \in [\![Q_A^{pre}]\!]\}$. Then, the reliability factor for the precondition is $[\![\mathcal{R}^*(Q_A^{pre})]\!](\varepsilon, \varphi, \theta, \upsilon) = \sum_{\varepsilon_a \in ins(\varepsilon)} \varphi(\varepsilon_a)$.
- **Approximate output environments:** For the instructions s and environment ε, $outs(\varepsilon) = \{\varepsilon'_a \mid \langle s, \varepsilon\rangle \underset{0_\gamma, \psi, 0_\xi}{\Longrightarrow} (\varepsilon', \tau, 1) \wedge (\varepsilon', \varepsilon'_a, \theta, \upsilon) \in [\![Q_A^{post}]\!]\}$. Then, the postcondition's reliability factor is $[\![\mathcal{R}^*(Q_A^{post})]\!](\varepsilon', \varphi', \theta, \upsilon) = \sum_{\varepsilon'_a \in outs(\varepsilon)} \varphi(\varepsilon'_a)$.
- **Fully reliable execution summaries:** For the instructions s and environment ε, $ces(\varepsilon) = \{(\varepsilon_a, \varepsilon'_a) \mid (\varepsilon, \varepsilon_a, \theta, \upsilon) \in [\![Q_A^{pre}]\!] \wedge \langle s, \varepsilon_a\rangle \underset{\gamma[\theta], \psi, \xi[\theta]}{\Longrightarrow} (\varepsilon'_a, \tau, p_a) \wedge \mathsf{correct}(\tau)\}$, contains the corresponding starting and final approximate environments, ε_a and ε'_a, respectively, for the approximate execution in which all operations executed correctly, denoted with the predicate correct on the trace τ.

Proof of Lemma 3. We divide the proof in several steps:

Step 1 *(Initial Reliability)*: From the definition of reliability factors:

$$[\![\rho_{add}^{\ell} \cdot \mathcal{R}^*(Q_A^{pre})]\!](\varepsilon, \varphi, \theta, \upsilon) = \rho_{add}^{\theta(\ell)} \cdot \sum_{\varepsilon_a \in ins(\varepsilon)} \varphi(\varepsilon_a)$$

We can represent the sum on the right-hand side with the elements from the set $ces(\varepsilon)$. In particular, the first projection of the set $ces(\varepsilon)$ is the set of initial approximate environments that correspond to ε. Note that this set is equal to $ins(\varepsilon)$.

Since `add` is a single instruction, then the probability p_a of the transition $(\varepsilon_a, p_a, \varepsilon'_a) \in ces(\varepsilon)$ is equal to 1 if the instruction is exact (i.e., $\theta(\ell) = 0$) or ρ_{add} if the instruction is approximate (i.e., $\theta(\ell) = 1$). Moreover, from the definition of the paired execution, this product is equal to $\varphi'(\varepsilon'_a)$. Therefore, we can represent the previous sum equivalently as:

$$\rho_{add}^{\theta(\ell)} \cdot \sum_{\varepsilon_a \in ins} \varphi(\varepsilon_a) = \sum_{\epsilon \in ces(\varepsilon)} \varphi(\pi_1(\epsilon)) \cdot \rho_{add}^{\theta(\ell)} = \sum_{\varepsilon'_a \in \pi_2(ces(\varepsilon))} \varphi'(\varepsilon'_a)$$

Step 2 *(Relation Between Reliability Factors)*: We relate the previous equality that follows from the precondition Q_A^{pre} and the execution with the postcondition Q_A^{post} via the inequality on the generalized joint reliability factors

$$\sum_{\varepsilon'_a \in \pi_2(ces(\varepsilon))} \varphi'(\varepsilon'_a) \leq \sum_{\varepsilon'_a \in outs(\varepsilon)} \varphi'(\varepsilon'_a) = [\![\mathcal{R}^*(Q_A^{post})]\!](\varepsilon', \varphi', \theta, \upsilon) \tag{5.9}$$

where, in the case of r = `add` r_1 r_2, the projection of *ces* is:

$$\pi_2(ces(\varepsilon)) = \{\varepsilon'_a \mid (\varepsilon, \varepsilon_a, \theta, \upsilon) \in [\![Q_A^{pre}]\!] \wedge \langle r \texttt{ = add } r_1\ r_2, \varepsilon_a\rangle \xrightarrow[\gamma[\theta],\psi,\xi[\theta]]{C,p} \langle \cdot, \varepsilon'_a\rangle\},$$

Since the set $ces(\varepsilon)$ contains the executions in which the addition executes *correctly*, for each starting approximate environment ε_a corresponding to ε, the computation produces a unique resulting environment ε'_a.

The only part of the environment that is modified by this execution is the value of the register r (and the values of the registers r_1, r_2 and the rest of the program state remains the same). However, the predicate Q_A^{pre} does not reference the register r (because of the substitution $AE_{\psi,\mathcal{I}}(r$ = `add` r_1 $r_2)/\Delta(r)$ and because the translation in Section 5.1.2 does not alias r with r_1 or r_2). Therefore Q_A^{pre} is also valid after the execution of the instruction. Based on this discussion, we have an equality:

$$\begin{aligned}\pi_2(ces(\varepsilon)) = \{\varepsilon'_a \mid \langle r \texttt{ = add } r_1\ r_2, \varepsilon\rangle &\xrightarrow[0_\gamma,\psi,0_\xi]{C,1} \langle \cdot, \varepsilon'\rangle \wedge \\ \langle r \texttt{ = add } r_1\ r_2, \varepsilon_a\rangle &\xrightarrow[\gamma[\theta],\psi,\xi[\theta]]{C,p} \langle \cdot, \varepsilon'_a\rangle \wedge (\varepsilon', \varepsilon'_a, \theta, \upsilon) \in [\![Q_A^{pre}]\!]\}.\end{aligned}$$

For the `add` instruction, the set *outs* is equal to

$$outs(\varepsilon) = \{\varepsilon'_a \mid \langle r \texttt{ = add } r_1\ r_2, \varepsilon\rangle \xrightarrow[0_\gamma,\psi,0_\xi]{C,1} \langle \cdot, \varepsilon'\rangle \wedge (\varepsilon', \varepsilon'_a, \theta, \upsilon) \in [\![Q_A^{post}]\!]\}.$$

To show that Inequality 5.9 holds, it is therefore sufficient to show that $\pi_2(ces(\varepsilon)) \subseteq outs(\varepsilon)$. By comparing the two sets, this condition is satisfied if $[\![Q_A^{pre}]\!] \subseteq [\![Q_A^{post}]\!]$, or in the different notation, $Q_A^{pre} \Rightarrow Q_A^{post}$.

Step 3 *(Compare Accuracy Precondition and Postcondition)*: To demonstrate this set inclusion, recall that from Proposition 2 we know that $\Delta(r) \leq AE_{\psi,\mathcal{I}}(r$ = `add` r_1 $r_2) = \Delta(r_1) + \Delta(r_2) + \ell \cdot m$ for the arithmetic instruction with reduced precision (m is the maximum error from Section 5.3.6). Recall that Q_A^{pre} and Q_A^{post} have the form

$$\begin{aligned}Q_A^{pre} &= d \geq d_x \cdot \Delta(r_1) + \Delta(r_2) + \ell \cdot m + R' \text{ and} \\ Q_A^{post} &= d \geq d_x \cdot \Delta(r) + R',\end{aligned}$$

where R' is the remaining reliability factor term that does not reference the register r.

Therefore, we can conclude that every time $\Delta(r_1) + \Delta(r_2) + \ell \cdot m \geq \Delta(r)$, and since d_x and R' are the same in both predicates, if Q_A^{pre} is valid, so is Q_A^{post} (i.e., $Q_A^{pre} \Rightarrow Q_A^{post}$). Note that the opposite does not need to be true, i.e., if Q_A^{post} satisfied, i.e. the function of $\Delta(r)$ respects the bound, it is possible that Q_A^{pre} is not satisfied, i.e. the function of $\Delta(r_1) + \Delta(r_2) + \ell \cdot m$ is greater than the maximum value that satisfies the predicate.

□

Below, we outline the steps for proving the soundness of the remaining statements in the intermediate language:

Other Arithmetic and Memory Instructions. We can use the same reasoning to prove the rule for the remaining arithmetic operations. The main difference in the proof is in Step 3, where for each operation we can use the discussion in Section 5.3.6. Likewise, we can apply this reasoning to the operations that move data between registers and memory.

Sequence of Instructions. To show that $\{C^*_{\psi,\mathcal{I}}(s_1, C^*_{\psi,\mathcal{I}}(s_2, Q))\}$ `s1; s2` $\{Q\}$, we start from an inductive hypothesis that, for arbitrary postconditions Q' and Q'', the triples $\{C^*_{\psi,\mathcal{I}}(s_1, Q'))\}$ `s1` $\{Q'\}$ and $\{C^*_{\psi,\mathcal{I}}(s_2, Q)\}$ `s2` $\{Q''\}$ are valid. Then, the statement follows from the fact that the precondition of the second statement and the postcondition of the first statement refer to the same program point, i.e., $Q' := C^*_{\psi,\mathcal{I}}(s_2, Q)$.

Conditionals. To show that $\{C^*_{\psi,\mathcal{I}}(s_1, Q^*) \wedge C^*_{\psi,\mathcal{I}}(s_2, Q^*)\}$ `if` r_c s_1 s_2 $\{A \leq \mathcal{R}^*(Q_A)\}$, where $Q^* := A \leq \mathcal{R}^*(Q_A \wedge 0 \geq \Delta(r_c))$, we note that 1) the reliability of the operands updated inside the conditional depends on the probability that the execution takes the same path and 2) the interval analysis rests on the fact that the approximation does not change the control flow of the computation. The accuracy predicate $0 \geq \Delta(r_c)$ ensures that the approximation does not change the control-flow in the conditional. Because of the additional accuracy predicate, $Q^* \Rightarrow Q$, and therefore, the generated precondition is also valid for the original postcondition Q. Since the register r_c is not assigned to within the branches, the analysis of the branches does not change the predicate. From the inductive hypothesis, the analysis of each branch generates a sound precondition, and their conjunction (given the postcondition Q^*) ensures that the precondition for the conditional statement satisfies the precondition of the branches.

5.4 Energy Objective Construction

The objective of the optimization is to minimize the energy consumption of the unreliable computation, as a function of the configuration θ. To approximate this optimization objective, we consider a set of traces of the original program. We now define a set of functions that operate on these traces and give an estimate of the energy consumption of the unreliable program executions.

The approximate hardware model presents relative savings of operations and memories (e.g., approximate instruction has saves 20% of the energy of the exact operation), instead of unknown absolute savings (e.g., approximate instruction consumes 8 pJ instead of 10 pJ). Therefore, this section presents how to express the system energy consumption as a function of the relative operation and memory savings. We show how the analysis computes the expression for the relative energy consumption, which we denote as $\mathfrak{R}_{\psi}(\theta)$.

5.4.1 Absolute Energy Model

Energy Model Specification. We extend the hardware specification from Section 4.2.1 with the relative energy savings for each approximate arithmetic operation (for simplicity we use α_{int} for all integer and α_{fp} for all floating point instructions) and approximate memory and cache regions (α_{mem} and α_{cache}). The specification also contains the relative energy consumption of the system's components (μ_{CPU}, μ_{ALU}, and μ_{cache}) and relative instruction class energy rates (w_{fp} and w_{oi}).

Energy of System. We model the energy consumed by the system (E_{sys}) when executing a program under configuration θ with the combined energy used by the CPU and memory:

$$E_{sys}(\theta) = E_{CPU}(\theta) + E_{mem}(\theta).$$

Energy of CPU. We model the energy consumption of the CPU as the combined energy consumed by the ALU, cache, and the other on-chip components:

$$E_{CPU}(\theta) = E_{ALU}(\theta) + E_{cache}(\theta) + E_{other}.$$

Energy of ALU. Each instruction in the hardware specification may have a different energy consumption associated with it. However, for the purposes of our model, we let $\mathcal{E}_{int}$, $\mathcal{E}_{fp}$, $\mathcal{E}_{oi}$ be the average energy consumption (over a set of traces) of an ALU instruction, a FPU instruction, and other non-arithmetic instructions, respectively.

Using the instructions from the traces that represent kernel execution on representative inputs, we derive the following sets: *IntInst* is the set of labels of integer arithmetic instructions and *FPInst* is the set of labels of floating-point arithmetic instructions. For each instruction with a label ℓ, we also let n_ℓ denote the number of times the instruction executes for the set of inputs. Finally, let α_{int} and α_{fp} be the average savings (i.e., percentage reduction in energy consumption) from executing integer and floating-point instructions approximately, respectively. Then, the ALU's energy consumption is:

$$E_{int}(\theta) = \sum_{\ell \in IntInst} n_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{int}) \cdot \mathcal{E}_{int}$$
$$E_{fp}(\theta) = \sum_{\ell \in FPInst} n_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{fp}) \cdot \mathcal{E}_{fp}$$
$$E_{ALU}(\theta) = E_{int}(\theta) + E_{fp}(\theta) + n_{oi} \cdot \mathcal{E}_{oi}.$$

This model assumes that the instruction count in the approximate execution is approximately equal to the instruction count in the exact execution.

Memory Energy. We model the energy consumption of the system memory (i.e., DRAM) using an estimate of the average energy per second per byte of memory, $\mathcal{E}_{mem}$. Given the execution time of all kernel invocations, t, the savings associated with allocating data in approximate memory, α_{mem}, the size of allocated arrays, S_ℓ, and the configurations of array variables in the exact and approximate memories, $\theta(\ell)$, we model the energy consumption of the memory as follows:

$$E_{mem}(\theta) = t \cdot \mathcal{E}_{mem} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{mem}).$$

Cache Memory Energy. We model the energy consumption of cache cell, $\mathcal{E}_{cache}$, similarly. Let S_c be the size of the cache, α_{cache} the savings of approximate caches. In addition, we need to specify the strategy for determining the size of approximate caches. We analyze the strategy that scales the size of approximate caches proportional to the percentage of the size of the arrays allocated in the approximate main memory. If c_u is the maximum fraction of the approximate cache lines, the energy consumption of the cache is

$$E_{cache}(\theta) = t \cdot \mathcal{E}_{cache} \cdot S_c \cdot (1 - c_u \cdot \frac{\sum_\ell S_\ell \theta(\ell)}{\sum_\ell S_\ell} \cdot \alpha_{cache}).$$

5.4.2 Relative Energy Model

While the energy model equations from Section 5.4.1 capture basic properties of energy consumption, the models rely on several hardware design specific parameters, such as the average energy of instructions.

However, we can use these equations to derive a numerical optimization problem that instead uses cross-design parameters (such as the relative energy between instruction classes and the average savings for each instruction) to optimize the energy consumption of the program relative to an exact configuration of the program, 0_θ (Section 4.2.4). For each energy consumption modeling function in the previous section we introduce a corresponding function that implicitly takes 0_θ as its parameter. For example, for the energy consumption of the system, we let $E_{sys} \equiv E_{sys}(0_\theta)$.

System Relative Energy. The energy model contains a parameter that specifies the relative portion of energy consumed by the CPU versus memory, μ_{CPU}. Using this parameter, we derive the relative system energy consumption as follows:

$$\boxed{\mathfrak{R}_\psi(\theta) = \frac{E_{sys}(\theta)}{E_{sys}}} = \frac{E_{CPU}(\theta) + E_{mem}(\theta)}{E_{CPU} + E_{mem}} =$$

$$= \frac{E_{CPU}}{E_{CPU}} \cdot \frac{E_{CPU}(\theta)}{E_{CPU} + E_{mem}} + \frac{E_{mem}}{E_{mem}} \cdot \frac{E_{mem}(\theta)}{E_{CPU} + E_{mem}} =$$

$$= \frac{E_{CPU}}{E_{CPU} + E_{mem}} \cdot \frac{E_{CPU}(\theta)}{E_{CPU}} + \frac{E_{mem}}{E_{mem}} \cdot \frac{E_{mem}}{E_{CPU} + E_{mem}} =$$

$$= \mu_{CPU} \cdot \frac{E_{CPU}(\theta)}{E_{CPU}} + (1 - \mu_{CPU}) \cdot \frac{E_{mem}(\theta)}{E_{mem}}.$$

CPU Relative Energy. The energy model contains a parameter that specifies the relative portion of energy consumed by the ALU, μ_{ALU}, cache, μ_{cache}, and other components $\mu_{other} = 1 - \mu_{ALU} - \mu_{cache}$. We can then derive the relative CPU energy consumption similarly to that for the whole system:

$$\frac{E_{CPU}(\theta)}{E_{CPU}} = \mu_{ALU} \cdot \frac{E_{ALU}(\theta)}{E_{ALU}} + \mu_{cache} \cdot \frac{E_{cache}(\theta)}{E_{cache}} + \mu_{other}.$$

ALU Relative Energy. We apply similar reasoning to derive the relative energy consumption of the ALU:

$$\frac{E_{ALU}(\theta)}{E_{ALU}} = \mu_{int} \cdot \frac{E_{int}(\theta)}{E_{int}} + \mu_{fp} \cdot \frac{E_{fp}(\theta)}{E_{fp}} + \mu_{oi}.$$

The coefficients μ_{int}, μ_{fp}, and μ_{oi} are computed from the execution counts of each instruction class (n_{int}, n_{fp}, and n_{oi}) and the relative energy consumption rates of each class with respect to that of integer instructions (w_{fp} and w_{oi}). For example, if we let w_{fp} be the ratio of energy consumption between floating point instructions and integer instructions (i.e, $w_{fp} = \frac{\mathcal{E}_{fp}}{\mathcal{E}_{int}}$), then $\mu_{fp} = \frac{w_{fp} \cdot n_{fp}}{n_{int} + w_{fp} \cdot n_{fp} + w_{oi} \cdot n_{oi}}$.

Memory And Cache Relative Energy. Applying similar reasoning to the memory subsystem yields the following:

$$\frac{E_{mem}(\theta)}{E_{mem}} = \frac{1}{H} \cdot \frac{t'}{t} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{mem})$$
$$\frac{E_{cache}(\theta)}{E_{cache}} = \frac{1}{H} \cdot \frac{t'}{t} \cdot \sum_{\ell \in ArrParams} S_\ell \cdot (1 - c_u \cdot \theta(\ell) \cdot \alpha_{cache}),$$

where $H = \sum_\ell S_\ell$ is the total size of heap data. The execution time ratio t'/t denotes possibly different execution time of the approximate program. One can use the results of reliability profiling to estimate this ratio.

Relative Energy for Multiple Inputs. The relative energy consumption for multiple inputs are the average of the relative energy consumption $E_{sys}(\theta)/E_{sys}$ for each input. Since this quantity is a sum of relative energy consumption of the components (CPU, ALU operations, and memories), the analysis computes and assigns these average relative energy consumption to each operation and variable label.

5.5 Final Optimization Problem Statement

We now state the optimization problem for a kernel computation:

$$
\begin{array}{lll}
\textbf{\textit{Minimize:}} & \mathfrak{R}_{\psi}(\theta) & \\
\textbf{\textit{Constraints:}} & \log(\rho_{spec,i}) - \log(\rho_i) \leq \sum_k \theta(\ell_{k_i}) \cdot \log(\rho_{k_i}) & \\
 & d_{spec,i} - d_i^* \geq \sum_k d_{k_i} \cdot \theta(\ell_{k_i}) & \forall i \\
\textbf{\textit{Variables:}} & \theta(\ell_1), \ldots, \theta(\ell_n) \in \{0, 1\} &
\end{array}
$$

The decision variables $\theta(\ell_1), \ldots, \theta(\ell_n)$ are the configuration kinds of arithmetic instructions and array variables. Since they are integers, the optimization problem belongs to the class of integer linear programs. The index i iterates over all constraints generated by the reliability and accuracy analyses. The index k iterates over the sequences of the candidate approximate instructions in the constraints.

Complexity. The number of constraints for a single program path is linearly proportional to the number of kernel outputs (the return value and the array parameters). The number of paths that Chisel's precondition generator produces is in the worst case exponential in the number of control flow divergence points. However, in practice, one can use the simplification procedure from [19, Section 5.4], which can identify most of the path predicates as redundant and remove them during the analysis. Out of the remaining predicates, Chisel can immediately solve those that involve only numerical parameters and pass only the optimization constraints with kind configurations to the optimization solver.

The number of decision variables is proportional to the number of instructions and array parameters in a kernel. In general, integer linear programming is NP complete with respect to the number of decision variables. However, existing solvers can successfully and efficiently solve many classes of integer linear programs with hundreds of variables.

We describe two techniques that can reduce the size of the generated optimization problem. First, the precondition generator can create constraints at coarser granularities. For example, a single decision variable may represent program statements, basic blocks, or loop bodies. (Section 6.9.1). Second, Chisel can separately optimize the invoked functions that implement hierarchically structured kernels (Section 6.9.2).

5.6 Discussion

This chapter presented the analysis of reliability, accuracy, and energy savings of the approximate kernel computations. This section discusses computational patterns with approximate kernels that are amenable to Chisel's optimization and challenges when extending the analysis from kernel computations to full programs.

5.6.1 Computational Patterns with Approximate Kernels

We anticipate that Chisel's optimization is particularly useful for the applications that naturally expose approximate kernels. Examples of such application domains include machine learning, multimedia, information retrieval, scientific, and financial analysis applications [19, 25, 27, 113]. Our previous work [19] and applications that we evaluate in Chapter 6 represent typical examples of computations that expose approximate kernels. These computations fall in two general patterns – approximate computations that naturally tolerate errors and checkable computations that check for and correct errors.

Approximate Computations. An approximate computation can often acceptably tolerate occasional errors in its execution or the data that it operates on. These computations are often implemented as loops that iterate over data. Chisel is well-suited for optimizing approximate kernels that can represent a body of such a loop.

Computational patterns that calculate order statistics, such as minimums/maximums or medians of elements computed by an approximate kernel often tolerate occasional unbounded errors. In case of error, these approximate computations may not produce an exact result, but will typically return one of the similar, but slightly less desirable results. Example of such a computation can be Argmin pattern from Section 2.4.2. Our previous work [19] shows that approximating motion estimation computation in x264 (that produces an exact minimum block with probability at least 0.98), results in small overall errors (less than 1% quality loss of the final video).

Computational patterns that calculate an average or a sum of elements computed by an approximate kernel often tolerate small bounded errors, even if the errors occur most of the time. While the errors of the individual elements often cancel out, even if the errors introduce a systematic bias in the computation's result, the approximate computation still preserves the result's central tendency. However, an occasional unbounded error can significantly affect the result of such computation. In such cases, a developer may augment the kernel with a *sanity checker* (which we describe below) to ensure that the result is within the expected range.

Computational patterns that represent a map computation, i.e., loops that independently run an approximate kernel for each input can often tolerate both the occasional unbounded or bounded error. For instance, a single execution of an approximate kernel in the scale example (Section 4.1) computes a single pixel value. A human eye can often tolerate such inaccuracies. If the unbounded error is unacceptable, a developer may provide sanity checks.

Computational patterns that represent an iterative computation, i.e., loops whose each iteration refines the result computed by an approximate kernel until the loop exit criterion is met, can often tolerate errors at the expense of a greater number of iterations or producing a less accurate output [71]. A developer can derive the kernel's specification from the estimate of how many more iterations will typically the approximate iterative computation execute.

Checkable Computations. A developer can augment an approximate computation with an efficient *sanity checker* that dynamically validates the correctness of the computations result. If a checker detects an error, then it reexecutes the approximate kernel. While sanity checkers are kernel-specific, we can classify them as full and partial correctness checkers.

A *full correctness checker* detects whether the result of an approximate kernel is correct. Examples of such checks can be in numerical computations (e.g., Newton's method example from [19]) or computations that solve NP-hard problems (e.g., solving SAT problem for a logical formula is computationally hard, but checking whether some assignment of logical variables is correct is computationally much simpler).

A *partial correctness checker* detects whether the result of an approximate kernel satisfies some important integrity property of the computation. For instance, a distance metric should not return a negative value, or an index of a most desirable element in an array must point within the array's bounds. In addition to manually implemented checkers, researchers have recently studied design of automatic outlier detectors [2] that can learn the typical output ranges and identify if a computation produces a likely incorrect result.

5.6.2 Limitations of Chisel's Optimization

While analysis-based accuracy-aware optimization can, in principle, optimize full programs, we anticipate that such analysis will often produce unacceptably low reliability and accuracy expressions. Below we discuss limitations and sources of imprecision of the analyses.

Monotonicity of Reliability/Accuracy Analysis. The expressions that the reliability and accuracy analyses generate are *monotonic* in a sense that each additional analyzed operation decreases the reliability of the computation and increases the absolute error, respectively. Therefore, applying this reasoning for long chains of instructions (such as a full program) will likely result in unacceptable reliability.

To alleviate this problem, a developer can use approximate kernels as parts of specific computational patterns, such as those that we described in Section 5.6.1. By using properties of such patterns, a developer may find accuracy specifications for kernels that satisfy a program-level sensitivity metric (e.g., using the analytic approach presented in Section 4.1.2). In addition, a developer may identify and optimize multiple kernels in a program. We discuss optimizing multiple kernels in Section 6.9.8.

Imprecision of Analyses. While the reliability and accuracy analyses produce sound error frequency and magnitude bounds, they can suffer from imprecision that cause the analysis to produce a more conservative reliability/accuracy expressions and thus miss some optimization opportunities.

Reliability analysis computes a probability that all instructions in a sequence execute correctly. It is imprecise in the sense that it does not account for the probability that the computation can return to the exact execution after experiencing multiple faults. However, this design choice makes the analysis computationally tractable (specifically computing a lower bound on $\mathcal{R}(O)$). In addition, reliability predicate generated by this analysis is sound for a relaxed fault model in which, after the first failed instruction, the subsequent instructions can experience correlated errors (e.g., with higher failure probability).

Accuracy analysis computes a magnitude of absolute error of approximate arithmetic operations. It has three main sources of imprecision. First, the underlying interval analysis computes imprecise intervals in the presence of control flow. Second, the error propagation analysis computes the sensitivity coefficients assuming that all of the kernel's operations are approximate. Third, the error magnitude bounds depend on the input intervals specified by a developer. To make the analysis more precise, a developer may run Chisel on several input intervals to produce multiple kernel implementations specialized for these intervals.

Both reliability and accuracy analyses are imprecise when analyzing arrays. Specifically, the analysis of arrays abstracts away accesses to different indices. The analyses treat all updates to an array as an update to a single shared element.

Energy analysis computes an expression that characterizes relative energy savings. It has two main sources of imprecision. First, it operates on a set of specific inputs (which affects the collected execution traces, number of executed iterations of bounded loops, and the size of the arrays) and can therefore overfit to those inputs. Second, it assumes that control flow deviations have a small impact on the kernel's execution time and ignores such time/energy savings differences caused by divergent control flow (we discuss in Section 6.9.5 how to penalize instructions that can impact control flow).

5.6.3 From Kernel Optimization to Full Program Optimization

While Chisel's optimization algorithm provides rigorous optimization of approximate kernels, many times these kernels are parts of larger, more complex programs, for which the full program analysis is too imprecise or intractable. In such cases,

When approximate kernels are parts of larger, more complex programs, for which the full program analysis is too imprecise, a developer can use sensitivity profiling to 1) help identify kernel computations and 2) derive the kernel-level accuracy specifications that likely satisfy the application-level sensitivity metric. Therefore, Chisel enables the developer to expose approximate kernels, and separate them from the remaining outer ("connective") code, which transfers data between kernel executions, makes decisions based on approximate kernel's results, or performs some auxiliary computation.

Chisel's core algorithm optimizes time- and energy-consuming kernels for which a developer provides a formal specification of the input intervals and output accuracy. Chisel uses static analysis to compute reliability and accuracy constraints in the underlying integer linear program. With this approach, Chisel can find the versions of the kernel that maximize savings, while satisfying the developer's accuracy specification. This optimization with exposed kernels resembles the scenario we described in Figure 1.1b.

Since approximate kernels have a specific function within the program, from the perspective of the outer code, the kernel specifications quantify accuracy loss, which indicates how a kernel's result may affect the outer computation. For example, Section 5.6.1 discusses how a developer can reason about several larger computational patterns that interact with approximate kernels. Therefore, exposing the kernel and its explicit specification allows the developer of the outer code to reason about approximation at the level of the kernel, rather than at the level of its individual instructions. This separation with a defined specification can help both during development and debugging of programs.

From the perspective of the kernel computation, the specification abstracts away the complexity of the outer computation and allows Chisel optimization system to focus full optimization effort on the kernel computation, while providing rigorous guarantees for the kernel's accuracy specification. Chisel's automatic and rigorous optimization provides a developer with the flexibility to seamlessly change the specification of the kernel computations if the application's requirements change, port the computation to another approximate hardware device with a different hardware specification, or dynamically adapt a program execution to run an alternative approximate versions to adjust to the fluctuations in the program's operational environment.

6 Evaluation and Extensions of Chisel Optimization Algorithm

We evaluate Chisel on several applications over a parameterized space of approximate hardware designs. Our evaluation consists of the following parts:

Sensitivity Profiling. We present how sensitivity profiling can help developers effectively identify an appropriate reliability specification for an application.

Optimization Problem Size. We present statistics that characterize the size of Chisel's optimization problem.

Energy Savings. We present the percentage of potential energy savings that Chisel found in the approximate kernels.

Output Quality. We present the resulting end-to-end sensitivity metric for the execution of the synthesized approximate benchmarks on a set of test inputs.

6.1 Chisel Implementation

We have implemented Chisel using OCaml. To generate the optimization problem, Chisel's optimizer implements the reliability analysis (Section 5.2), the accuracy analysis (Section 5.3) and the energy analysis (Section 5.4). To solve the optimization problem, we use Gurobi mixed integer programming solver [48]. Finally, the transformation pass uses the optimization problem solution to generate a source code of the kernel with Rely annotations for approximate instructions and memory annotations.

The framework also contains several helper passes. The C-translation pass produces a (fully reliable) C program that allows executing the kernel written in the Rely language.

A sensitivity profiler transforms the kernel to generate the computation that will generate the correct result only with a specified probability. It uses the C-translation pass to generate

Approximation	Mild		Medium		Aggressive	
	Failure Rate	Savings	Failure Rate	Savings	Failure Rate	Savings
Arithmetic Operation Timing Errors	10^{-6}	12%	10^{-4}	22%	10^{-2}	30%
Cache Read Upset	$10^{-16.7}$	70%	$10^{-7.4}$	80%	10^{-3}	90%
Cache Write Upset	$10^{-5.59}$		$10^{-4.94}$		10^{-3}	
DRAM refresh error	10^{-9}	17%	10^{-5}	22%	10^{-3}	24%

Table 6.1: Approximate Hardware Configurations and Operation Failure Rates from [105]

the exact implementation of the kernel. It also implements a search algorithm to find the reliability specification for the function that corresponds to the developer's accuracy target.

A trace profiler generates an instrumented C program from the kernel. The instrumentation collects traces as sequences of assembly instructions from Section 4.2. These frequencies are used to compute the frequencies of instructions (n_{int}, n_{fp}, and n_{oi}) in the energy objective.

The framework also contains a fault injection pass that simulates the execution of the computation on approximate hardware. This pass injects random errors at the locations of the approximate arithmetic instructions and memory instructions operating on approximate data (given the specification of approximate hardware). It collects execution statistics of the approximate computation with errors (including the quality of the output and the frequencies of approximate instructions executed the traces used to estimate savings).

6.2 Hardware Reliability and Energy Specifications

We use the reliability and energy specifications for approximate hardware initially presented in [105, Table 2] to instantiate our approximate hardware specification, ψ. We reproduce this data in Table 6.1. It defines three configurations, denoted as *mild*, *medium* and *aggressive*, for arithmetic instructions, caches, and main memories respectively. We consider only the unreliable arithmetic operations (that produce the correct results with specified probability) and unreliable memories.

System Parameters. To compute the overall system savings (Section 5.4.2), we use the server configuration parameters specified in [105, Section 5.4]: CPU consumes $\mu_{CPU} = 55\%$ of energy and the main memory consumes the remaining 45%; the ALU consumes $\mu_{ALU} = 65\%$ of CPU's energy and the cache consumes the remaining $\mu_{cache} = 35\%$ energy.

The sizes of the reliable and approximate regions of the main memory are determined before the execution of the kernel computations and remain fixed until all kernel computations finish. We assume that the capacity of the approximate region of the cache (that can store approximate heap data) is twice that of the reliable cache that contains instructions and reliable data, and therefore $c_u = 67\%$.

Error Model. The error injection pass and its runtime insert faults in the synthesized computation with the frequency specified by the hardware specification. For integer and floating point ALU operations, the error model returns a fully random result (as in [105]). For read and write memory errors, the error model flips from one (with highest probability) up to three bits (with lowest probability) in the word.

6.3 Benchmarks

We implemented a set of benchmarks from several application domains. The benchmarks were selected because they tolerate some amount of error in the output:

- **Scale.** Scales an image by a factor provided by the user. The kernel computes the output pixel value by interpolating over neighboring source image pixels.
- **Discrete Cosine Transform (DCT).** A compression algorithm used in various lossy image and audio compression methods. The kernel computes a frequency-domain coefficient of an 8x8 image block.
- **Inverse Discrete Cosine Transform (IDCT).** Reconstructs an image from the coefficients generated by DCT. The kernel reconstructs a single pixel from the frequency domain grid.
- **Black-Scholes.** Computes the price of a portfolio of European Put and Call options using the analytical Black-Scholes formula. The kernel calculates the price of a single option. Our implementation is derived from the benchmark from the PARSEC benchmark suite [118].
- **Successive Over-relaxation (SOR).** The Jacobi SOR computation is a part of various partial differential equation solvers. The kernel averages the neighboring matrix cells computed in the previous iteration. It is derived from the benchmark from the SciMark 2 suite [119].

Benchmark	Size (LoC)	Kernel (LoC)	Time in Kernel %	Array Parameter Count / Heap %	Representative Inputs (Profile/Test)
scale	218	88	93.43%	2 / 99%	13 (5/8)
dct	532	62	99.20%	2 / 98%	13 (5/8)
idct	532	93	98.86%	2 / 98%	9 (3/6)
blackscholes	494	143	65.11%	6 / 84.4%	24 (8/16)
sor	173	23	82.30%	1 / 99%	20 (6/14)

Table 6.2: Benchmark Description

Benchmark	Sanity Test	Sensitivity Metric
scale	×	Peak Signal-to-Noise Ratio
dct	×	Peak Signal-to-Noise Ratio
idct	×	Peak Signal-to-Noise Ratio
blackscholes	✓	Relative Portfolio Difference
sor	✓	Average Relative Difference

Table 6.3: Description of Benchmark's Accuracy Metric

Table 6.2 presents an overview of the benchmark computations. For each computation, Column 2 ("Size") presents the number of lines of code of the benchmark computation. Column 3 ("Kernel") presents the number of lines of kernel computation that is a candidate for optimization. Column 4 ("Time in Kernel %") presents the percentage of instructions that the execution spends in the kernel computation. Column 5 ("Array Parameter Count/Heap %") presents the number of array arguments and the percentage of heap allocated space that these variables occupy. Column 6 ("Representative Inputs") presents the number of representative inputs collected for each computation.

Table 6.3 presents the application's accuracy requirement. Column 2 ("Sanity Test") presents whether the computation contains a sanity test that ensures the integrity of its result. Column 3 ("Sensitivity Metric") presents the sensitivity metric of the computation.

Representative Inputs. For each benchmark, we have selected several representative inputs. The analysis uses a subset of these inputs (designated as "Profile") to obtain the estimates of the instruction mixes and construct the objective function of the optimization problem. We use the remaining inputs (designated as "Test") to evaluate the synthesized approximate computation.

Sensitivity Metrics. For the three image processing benchmarks (Scale, DCT, and IDCT) we use peak signal to noise ratio between images produced by the original and the

Benchmark	Reliability Bound	Sensitivity metric	
		Average	Conservative
scale	0.995	30.93 ± 0.95 dB	23.01 dB
dct	0.99992	27.74 ± 1.32 dB	22.91 dB
idct	0.992	27.44 ± 0.49 dB	20.96 dB
blackscholes	0.999	0.005 ± 0.0005	0.05
sor	0.995	0.058 ± 0.034	$\geq$ 1.0

Table 6.4: Software Specification PSNR and Sensitivity Profiling

synthesized versions of the benchmark. Specifically for DCT, the sensitivity metric converts the image from the frequency domain and computes the PSNR on the resulting image.

For Blackscholes, we have used the relative difference between the sum of the absolute errors between the option prices and the absolute value of the price of the portfolio (the sum of all option values returned by the fully accurate program). For SOR, the sensitivity metric is the average relative difference between the elements of the output matrix.

Sanity Tests. Two of the benchmark computations have built-in sanity test computations that ensure that the intermediate or final results of the computation fall within specific intervals. These computations typically execute for only a small fraction of the total execution time. The Blackscholes sanity test uses a *no-arbitrage* bound [12] on the price of each option to filter out executions that produce option prices that violate basic properties of the Black-Scholes model. The SOR benchmark checks whether the computed average is between the minimum and maximum array value.

If the sanity test computation fails, the approximate computation may skip updating the result (as in SOR), or reexecute the computation (as in Blackscholes). In the case of reexecution, the overall savings are scaled by the additional execution time of the kernel computation.

6.4 Sensitivity Profiling Results

To find reliability specifications for the benchmark applications, the sensitivity profiler relates the reliability degradation of a kernel computation with the benchmark's end-to-end sensitivity metric.

Methodology. For each profiling input, we perform 100 fault injection experiments. As in Section 4.1.2, we use the sensitivity profiler to compute the average sensitivity metric value (over the space of possible injected faults) for multiple reliability bounds using a

Benchmark	Optimization Variables	Reliability Constraints
scale	147	4
dct	121	1
idct	104	1
blackscholes	77	2
sor	36	1

Table 6.5: Optimization Problem Statistics

developer-provided sensitivity testing procedure. For each benchmark, we select one reliability bound that yields an acceptable sensitivity metric. We also analytically derive conservative estimates of the average sensitivity metric.

Table 6.4 presents the final reliability specifications for the benchmarks. Column 2 presents the reliability bound. Column 3 presents the average metric obtained from sensitivity testing. Column 4 presents the analytic conservative lower bound on the average sensitivity metric.

Image Benchmarks. For Scale and IDCT, the sensitivity testing procedure (like the one from Section 4.1.2) modifies a single pixel. For DCT, the sensitivity testing procedure changes a single coefficient in the 8x8 DCT matrix. To compute the lower bound on the average PSNR, we use an analytical expression from Section 4.1.2. Note that DCT has smaller average PSNR than the other two benchmarks, as a single incorrectly computed coefficient can make 64 pixels in the final image incorrect.

Blackscholes. The sensitivity testing procedure conservatively estimates the error at the end of the computation by returning either the upper or the lower no-arbitrage bound (whichever is more distant from the exact option value). For reliability bound 0.999, the average absolute error is \$150.6 (±\$1.63), while the average value of the portfolio is \$28361.4. Therefore, the relative error of the portfolio price is approximately 0.50%. To derive a conservative analytic expression for the deviation, we use the no-arbitrage bound formula, while assuming that the price of the portfolio is at least \$4000 (e.g., each option in a portfolio with 4K options is worth at least a dollar) and the strike price is less than \$200.

SOR. For SOR, the sensitivity testing strategy returns a random value within the typical range of the input data. Since the computation performs multiple updates to the elements of the input matrix, the worst-case relative error typically exceeds 100%.

6.5 Optimization Problem Solving Results

Chisel's optimization algorithm constructs the optimization problem and calls the Gurobi solver. Table 6.5 presents for each benchmark the number of variables (Column 2) and the number of constraints (Column 3) constructed by Chisel. For each of these problems, Gurobi took less than a second to find the optimal solution (subject to optimality tolerance bound 10^{-9}) on an 8-core Intel Xeon E5520 with 16 GB of RAM.

6.6 Energy Savings Results

We next present the potential savings that Chisel's optimization uncovered. We relate the savings obtained for the traces of profiled inputs to 1) the maximum possible savings when the reliability bound is 0.0 and 2) the savings for the previously unseen test inputs.

Methodology. To get the statistics on the approximate execution of the benchmarks, we run the version of the benchmark transformed using the fault injection pass. We ran the benchmark 100 times for each test input. To estimate the energy savings, we use the instruction counts from the collected traces and the expressions derived in Section 5.4.

Results. Table 6.6 presents the system savings that Chisel's optimization algorithm finds for the kernel computations. For each benchmark, we present the target *reliability bound* that we set according to the exploration in Section 6.4. We also present the *potential savings*, which are the maximum possible savings when all instructions and memory regions have medium configuration (Table 6.1) and the result's reliability bound is 0.0 – so that all operations can be unreliable and all arrays can be stored in unreliable memory.

For each benchmark, Column 1 ("Hardware Configuration") presents the approximate hardware configuration. We represent the system configurations as triples of the form *CPU/Cache/Main*, denoting the reliability/saving configuration of CPU instructions, cache memories, and main memories, respectively. We use the letters "m" and "M"' to denote the mild and medium reliability/savings configuration of the system component from Table 6.1. We omit the aggressive configurations as they yield no savings or only small savings for the reliability bounds of our benchmarks. For instance, the configuration "M/m/M" denotes a medium configuration for CPU instructions, mild configuration for the cache memory, and medium configuration for the main memory. The column "Profile" contains the savings that Chisel finds for the inputs used in sensitivity profiling. The column "Test" contains savings computed from the traces of inputs not used during sensitivity profiling.

Benchmark: scale
Reliability Bound: 0.995
Potential Savings: 20.28%

Hardware Configuration	Energy Savings		Sensitivity Metric for Test Inputs
	Profile	Test	
m/m/m	14.11%	14.16%	44.79 ± 2.51
M/m/m	14.22%	14.28%	35.30 ± 1.95
M/M/m	15.39%	15.42%	34.07 ± 1.19
M/m/M	18.17%	18.20%	33.13 ± 1.33
M/M/M	19.35%	19.36%	32.31 ± 1.08

Benchmark: dct
Reliability Bound: 0.99992
Potential Savings: 20.09%

Hardware Configuration	Energy Savings		Sensitivity Metric for Test Inputs
	Profile	Test	
m/m/m	6.73%	6.72%	30.34 ± 3.84
M/m/m	6.73%	6.73%	30.37 ± 4.41
M/M/m	0.00%	–	–
M/m/M	8.72%	8.72%	29.76 ± 4.81
M/M/M	0.00%	–	–

Benchmark: idct
Reliability Bound: 0.992
Potential Savings: 19.96%

Hardware Configuration	Energy Savings		Sensitivity Metric for Test Inputs
	Profile	Test	
m/m/m	13.38%	13.38%	31.28 ± 0.80
M/m/m	13.40%	13.40%	30.45 ± 0.75
M/M/m	7.34%	7.34%	30.36 ± 0.19
M/m/M	8.70%	8.70%	30.36 ± 0.18
M/M/M	9.32%	9.32%	30.35 ± 0.20

Benchmark: blackscholes
Reliability Bound: 0.999
Potential Savings: 17.39%

Hardware Configuration	Energy Savings		Sensitivity Metric for Test Inputs
	Profile	Test	
m/m/m	9.87%	9.79%	0.0002 ± 0.00004
M/m/m	9.90%	9.81%	0.0006 ± 0.00008
M/M/m	5.38%	5.35%	0.0005 ± 0.00006
M/m/M	6.36%	6.32%	0.0005 ± 0.0008
M/M/M	4.40%	4.52%	0.0005 ± 0.0008

Benchmark: sor
Reliability Bound: 0.995
Potential Savings: 20.07%

Hardware Configuration	Energy Savings		Sensitivity Metric for Test Inputs
	Profile	Test	
m/m/m	14.52%	14.50%	0.029 ± 0.022
M/m/m	14.83%	14.87%	0.051 ± 0.032
M/M/m	16.07%	16.07%	0.046 ± 0.038
M/m/M	18.81%	18.70%	0.086 ± 0.090
M/M/M	19.83%	19.43%	0.080 ± 0.074

Table 6.6: Energy Savings and Sensitivity Metric Results (Configurations: 'm' denotes mild and 'M' medium CPU/Cache/Memory approximation). Shaded configuration yields maximum energy savings.

Overall, for these benchmarks and hardware specification, the majority of savings (over 95%) come from storing data in unreliable memories. For Scale and SOR, Chisel marks all array parameters and a significant portion of instructions as unreliable for the configuration "M/M/M". For Scale, the optimization achieves over 95% (19.35% compared to 20.28%) of the maximum savings. For SOR, it obtains more than 98% of the maximum savings.

In general, the hardware parameters affect the result that Chisel produces. For instance, Chisel cannot apply any approximation for the medium main memory configuration for DCT (which is the benchmark with the strictest reliability bound) – it produces a kernel in which all operations are reliable. However, for mild memory and cache configurations, the optimization can obtain up to 43% of the maximum possible savings.

For IDCT, Chisel obtains greater savings for mild ("m") configurations of the unreliable memories, because in this case both arrays (passed as the function parameters) can be allocated in the unreliable memory, to obtain 67% of the maximum possible savings. When the memory configurations are at the medium ("M") level, Chisel can place only one array parameter in unreliable memory.

For Blackscholes, Chisel also selects different combinations of unreliable input array parameters based on the configurations of the main and cache memories and exposes up to 57% of the maximum possible savings. Blackscholes reexecutes some of its computation (when detected by the sanity test), but this reexecution happens for only a small fraction of the options (less than 0.03% on average) and has a very small impact on program's execution time and energy consumption.

For all benchmarks, the energy savings obtained on the test inputs typically have a deviation less than 3% from the savings estimated on the profiling inputs.

6.7 Output Quality Results

We next present the end-to-end sensitivity metrics results for the executions of programs with synthesized kernels.

Methodology. We instrumented the unreliable operations selected by the optimizer and injected errors in their results according to the hardware specification and error model.

Results. Table 6.6 also presents the end-to-end sensitivity of the optimized benchmarks. Column 4 ("Sensitivity Metric for Test Inputs") presents the mean and the standard deviation of the distribution of the error metric. The number of faults per execution ranges from several (Blackscholes) to more than a thousand (DCT and IDCT).

The sensitivity metric of Scale, DCT, and IDCT is the average PSNR metric (higher value of PSNR means better accuracy). We note that the value of the metric for the synthesized computation is similar to the sensitivity profiling results (Table 6.4). The accuracy of Scale and IDCT increases for the mild configuration of arithmetical operations, as the frequency of faults and therefore the number of faulty pixels caused by computation decreases. The higher variance in DCT is caused by the inputs of a smaller size, where each fault can significantly impact PSNR.

The accuracy of blackscholes exceeds the accuracy predicted by the sensitivity testing (up to 0.06% on test inputs vs. 0.5% in sensitivity testing). The error injection results for SOR are less accurate than the sensitivity profiling results for medium main memory configurations (8.0% vs. 5.8%). We attribute this lower accuracy to the fact that the sensitivity profiling does not inject errors in the read-only edge elements of the input matrix.

6.8 Kernel Transformations

We now focus on the kernels that Chisel's optimization algorithm generated. For each benchmark, we examined the kernel with maximum energy savings. Appendix A presents the optimized programs that Chisel produced.

Scale. We discussed the transformation of scale's kernel in Section 4.1.3.

DCT. Chisel places the array that contains the pixels of the output image in the unreliable memory. All arithmetic operations remain reliable, as they all occur in a nested loop.

IDCT. Chisel places both arrays (these arrays contain the pixels of the source and output image) in unreliable memory. Chisel also selects 14% of the arithmetic instructions as unreliable. The instrumented instructions include those that affect the condition of one of the inner bounded loops. Since this loop executes at most 8 iterations (which is enforced by the language semantics), this transformation does not have a visible impact on the energy consumption of the kernel.

Blackscholes. Chisel places 5 out of 6 input arrays in unreliable memory. These arrays contain different input parameters for computing the blackscholes equation. In addition, Chisel selects 7% of the arithmetic operations as unreliable that satisfy the specification.

SOR. Chisel places the input array in unreliable memory and selects 82% of the arithmetic operations as unreliable. These unreliable instructions do not affect the control flow.

6.9 Chisel's Extensions

This section describes several directions for how to extend Chisel's algorithm to support a wider set of computations.

6.9.1 Operation Selection Granularity

When the number of decision variables in the optimization problem for a large kernel computation is too large to solve given the computational resources at-hand, a developer may instruct the optimizer to mark all instructions in a block of code with the same kind (i.e., all exact or all approximate). The optimization algorithm assigns a single label ℓ to all operations within this block of code. This approach reduces the number of decision variables and, therefore, the resources required to solve the optimization problem.

6.9.2 Function Calls

To analyze function calls, one can use multiple strategies. We outline three such strategies.

Inlining. Chisel's preprocessor inlines the body of the called function before the precondition generation analyses. The constraint generator will then assign different labels to instructions inlined at each call site. This approach provides the finest granularity, but for kernels with a large number of call sites may increase the solving time.

Multiple Existing Implementations. A called function f may have multiple implementations, each with its own reliability specification. The specification of each of the m implementations of f consists of the function's reliability specification $\rho_{f,i} \cdot \mathcal{R}(\cdot)$ and estimated energy savings $\alpha_{f,i}$.

For n calls to the function f in the kernel, the constraint generator specifies the labels $\ell_{f,1,1}, \ldots, \ell_{f,m,n}$. The reliability expression for a k-th call site becomes $\prod_i \rho_{f,i}^{\ell_{f,i,k}}$. The relative ALU energy consumption expression for the same call site is $\mu_{f,k} \cdot (1 - \sum_i \theta(\ell_{f,i.k}) \cdot \alpha_{f,i})$. A trace profiler can record the count of instructions that the exact computation spends in each called function to calculate the parameters $\mu_{f,k}$.

We also specify a constraint $\sum_{i=1}^{m} \theta(\ell_{f,i.k}) = 1$ for each call site to ensure that the optimization procedure selects exactly one of the alternative implementations of f.

Inferring Reliability Specification. Instead of selecting from one of the predefined reliability specifications, one can use the optimization procedure to find the acceptable

reliability degradation of the called function f that will satisfy the reliability specification of the caller function. The constraint generator can then be extended to directly model the logarithm of the reliability as a continuous decision variable $\rho'(\ell_f) \leq 0$ (ℓ_f is the label of f).

For the energy consumption expression, the optimization requires the developer to provide a function $\alpha_f(\rho'(\ell_f))$, which specifies a lower bound on the energy savings. To effectively use an optimization solver like Gurobi, this function is required to be linear (the optimization problem is a mixed integer linear program), or quadratic (the optimization problem is a mixed integer quadratic program).

6.9.3 Overhead of Operation Mode Switching

Some approximate architectures impose a performance penalty when switching between exact and approximate operation modes due to e.g. dynamic voltage or frequency scaling. Therefore, for these architectures it is beneficial to incorporate the cost of switching into the optimization problem. For example, the constraint generator can produce additional constraints that bound the total switching overhead [109].

To specify this additional constraint, we let ℓ_i and ℓ_{i+1} be the labels of two adjacent arithmetic instructions. Next, we define auxiliary counter variables $s_i \in \{0, 1\}$ such that

$$s_i \geq \theta(\ell_i) - \theta(\ell_{i+1}) \quad \wedge \quad -s_i \leq \theta(\ell_i) - \theta(\ell_{i+1}).$$

Finally, we specify the constraint $\sum_i s_i \leq B$ to limit the total number of mode changes to be below the bound B.

6.9.4 Array Index Computations and Control Flow

Instead of relying on support for failure-oblivious program execution (Section 4.2.3), Chisel can further constrain the set of optimized instructions to exclude instructions that compute array indices and/or affect the flow of control. To ensure that approximate computation does not affect an expression that computes an array index or a branch condition, a dependence analysis can compute the set of all instructions that contribute to the expression's value. Chisel then sets the labels of these instructions to zero to indicate that the instructions must be exact.

6.9.5 Energy Analysis and Control Flow

The analysis used to estimate energy savings (presented in Section 5.4) assumes that the control flow divergences caused by approximate instructions and data have a negligible effect on the program's execution time and energy consumption. This assumption is valid when the

branches of the computation have a similar number of instructions or when the probability of computing incorrectly an if/loop condition is very small. When this assumption is not a good approximation, it is possible to specify a penalty for operations and data that affect the control flow. This penalty is proportional to the number of times each such operation is executed.

To add the impact of divergence to arithmetic operations, we define the set *CF* that contains all arithmetic instructions, which may affect the computation's control flow. Then the auxiliary analysis $\text{TraceMax}_{\gamma,\psi}(\ell)$ can compute the maximum energy consumption (according to the hardware specification ψ) of the traces in the program γ that start from the instruction with label ℓ. Conceptually, to implement such function, one can define a version of dataflow dependency analysis that counts the number of instructions across the traces.

Then, we can specify the absolute energy consumption of arithmetic instructions as, e.g., $E_{int}(\theta) = \sum_{\ell \in IntInst} n_\ell \cdot (1 - \theta(\ell) \cdot \alpha_{int}) \cdot \mathcal{E}_{int} + \sum_{\ell \in CF} (1 - \rho_\ell) \cdot \theta(\ell) \cdot \text{TraceMax}_{\gamma,\psi}(\ell)$. This expression states that with probability $1 - \rho_\ell$, the computation has the potential to use maximum energy if it takes (the unlikely) sequence of instructions. We can similarly add the term in the expressions for the energy consumption of memories. In that case, the penalty term would add this time to the numerator t' of the fraction t'/t.

6.9.6 Hardware with Multiple Approximate Operation Specifications

To support hardware platform with arithmetic operations and memory regions with multiple reliability/savings specifications $(\rho_{op,i}, \alpha_{op,i})$, we can use an approach analogous to the one for functions with multiple implementations. Specifically, each arithmetic operation can be analyzed as one such function. Analogously, to specify one of k approximate memory regions for a parameter v, the generator defines the labels $\ell_{v,1}, \ldots, \ell_{v,k}$. It generates the reliability expression $\prod_i \rho_{mop,i}^{\ell_{v,i}}$ for each memory operation and the memory savings expression $\sum_i \theta(\ell_{v,i}) \cdot \alpha_{mem,i}$ for each array parameter. To select a single memory region, the generator produces the constraint $\sum_i \theta(\ell_{v,i}) = 1$.

6.9.7 Interval-Based Reliability Specifications

Some approximate hardware designs may operate unreliably only if data is in certain intervals (e.g., having long carry propagation in arithmetic operations for large numbers). The extended approximate hardware specification can then contain optional input ranges for which the operation can operate unreliably. Chisel's analysis can use the helper interval analysis (Section 5.3.5) to check whether the result of an arithmetic operation belongs to

the interval that may exhibit unreliable hardware operation, and only in that case add the reliability term ρ^{ℓ} for the operation.

6.9.8 Multiple Kernels

A program may contain multiple approximate kernels. To adapt Chisel's workflow, we consider following modifications to the reliability profiling and the optimization.

Reliability Profiling. The approximate execution of one kernel may affect the inputs and the execution of the other kernels. Therefore, to find the reliability specifications of multiple kernels, the reliability profiler enumerates parts of the induced multidimensional search space. First, the one-dimensional profiler (Section 4.1.2) finds the lower reliability bound of each kernel. Then, to find the configuration of kernel reliability specifications that yield an acceptably accurate result, the profiler can systematically explore the search space, e.g. using strategies analogous to those that find configurations of accuracy-aware program transformations [53, 75, 79, 113] from a finite set of possible configurations. The profiler then returns configurations that closely meet the accuracy target, ordered by the reliability of the most time-consuming kernels.

Optimization. The optimization algorithm for multiple kernels needs to consider only the allocation of arrays, since the ALU operations are independent between kernels.

The multiple kernel optimization operates in two main stages. In the first stage, it computes the energy savings of each individual kernel for all combinations of shared array variables. Conceptually, if the shared variables are labeled as $\ell_1, \ldots, \ell_k$, the optimization algorithm calls the basic optimization problems for all combinations of the kind configurations $\theta(\ell_1), \ldots, \theta(\ell_k)$, while pruning the search tree when the algorithm already identifies that a subset of labels cannot satisfy the reliability bound.

In the second stage, the analysis searches for the maximum joint savings of the combination of m kernels. It searches over the combination of individual kernel results for which all array parameters have the same kind configuration, i.e., $\theta^{(1)}(\ell_i) = \ldots = \theta^{(m)}(\ell_i)$ for each $i \in \{1, \ldots, k\}$. The algorithm returns the combination of kernels with maximum joint energy savings, which is a sum of the kernels' savings weighted by the fraction of their execution time. While, in general, the number of individual optimization problems may increase exponentially with the number of shared array variables k, this number is typically small and the search can remain tractable.

7 Related Work

This chapter presents related work in the areas of software- and hardware-based approximate computing. We also present relevant related work from the areas of probabilistic program analysis, analysis of analytic program properties, and approximate databases.

7.1 Compiler-Level Approximations

Researchers have developed many systems that apply accuracy-aware transformations to reduce the amount of energy and/or time required to execute computations on commodity (exact and fully reliable) hardware platforms. Approximate transformations, such as task skipping [22, 71, 95], loop perforation [78, 79, 113], approximate function substitution [6, 9, 102, 125] (which replaces the exact implementation of a function with its less accurate alternative), dynamic knobs [53] (which selects at runtime one of the several approximate versions of the computation based on the application's performance goal), early termination of barriers at parallel loops [96] (which results in skipping contributions from interrupted threads), reduction sampling [102, 125] (which selects only a random subset of inputs from program codes that implement reductions like summation or maximization), approximate parallelization with data races [75, 80], and tuning floating-point operations [100, 110] can all reduce the energy and/or time required to execute the computation.

Common to all these transformations is that they create new knobs that expose and control the tradeoff between accuracy and performance/energy. To automatically justify these transformations, researchers have proposed various techniques for 1) analyzing sensitivity of computations, 2) analyzing safety of transformations, and 3) navigating the tradeoff space to find knob configurations that result in profitable tradeoffs.

7.1.1 Sensitivity Analysis

A critical region of program code, when transformed, causes unacceptable program errors (such as crashing, becoming unresponsive, or producing inadequate output). An approximable region of program code, when transformed, only affects the accuracy of the compu-

tation. Researches presented dynamic techniques for identifying critical and approximable code regions [10, 20, 79, 98, 99]. Researchers also presented static analysis techniques that reason about the propagation of errors through computation [24, 25, 31].

Dynamic Sensitivity Analysis. Testing-based sensitivity analyses can identify critical regions of the program. In general, dynamic sensitivity analyses transform a program's code, change program's inputs, or change its execution environment. The analyses reason about the effect of these changes on the program's output and classify the code regions accordingly. For instance, Section 2.1.2 presents a dynamic analysis that identifies critical parts of the program by transforming a program using loop perforation. This approach extends our early work on Quality of service profiling [79], which transforms program using loop perforation to separate critical and approximable regions of programs.

Snap [20] combines input fuzzing with dynamic execution and influence tracing to quantitatively characterize the sensitivity of the computation to changes to the input. ASAC [99] characterizes the sensitivity of the computation to changes in the intermediate program data. Ringenburg et al. [98] present quality of result debugger techniques that track approximate dataflow and identify the contribution of each approximate operation and correlation between approximate operations on the result's quality. Bao et al. [10] use whitebox sampling to find discontinuities in numerical computations.

In Chapter 4, we presented Chisel's sensitivity profiling that quantitatively relates the rate of incorrect kernel results to the quality of the result that the program produces. It relies on the developer-provided noise models (that a developer can provide as sensitivity testing functions). Chisel's sensitivity profiling differs from previous techniques in the source of the noise (incorrect approximate kernel results as opposed to changes in the computation or inputs) and the goal of the analysis (obtaining a reliability specification, in addition to checking the sensitivity of the kernel to noise).

Static Techniques. Chaudhuri et al. [24, 25] present techniques that statically reason about the propagation of errors through a continuous computation. A computation is continuous if small numerical changes in its input correspond to the small numerical changes in its output. A computation is Lipschitz continuous if the maximum output changes are multiples of the numerical change in the input. It can be used to reason about accuracy-aware transformations [25].

Researchers have also proposed techniques for analyzing the worst-case behavior of numerical computations. Researchers in embedded systems have traditionally used rounding error analyses of numerical programs to derive the worst-case error bounds for reduced-bitwidth

floating point computations [45, 88]. Recent techniques such as [31] verify the precision of approximate numerical computation and compute the bounds on error propagation through non-linear computation. Chisel uses a standard interval analysis as a component to construct accuracy constraints (which select which arithmetic operations can be approximated) and reason simultaneously about the frequency and magnitude of error.

7.1.2 Safety Analysis

While dynamic sensitivity profiling techniques can help identify critical parts of the program by finding a single failing execution, they are insufficient to prove the absence of errors or incorrect outputs. Therefore, researchers have developed various techniques that let a developer specify important safety property (such as non-interference of approximate and exact code, pointer safety, or range of values that the computation produces) and verify that the transformations preserve these properties.

EnerJ [105] presents an information-flow type system that allows the developer to separate code and data in distinct approximate and exact regions of code. The type system ensures that approximate data cannot be used to compute exact results (except when specially requested by the developer). More recently, Accept [104] presents an implementation of this type system for C/C++. Also, FlexJava [90] automates a part of approximate operation annotation through type inference.

Carbin et al. [16] present a general framework for reasoning about arbitrary safety properties (and also worst-case error) of approximate programs. It operates within Coq, an interactive theorem prover. It represents accuracy-aware transformations with *relax* and *assert* statements, which specify variables affected by the transformation, and how the transformation affects the values of program's variables, respectively. The framework relates the execution of the original and approximate programs, through a deterministic version of the paired execution semantics (which inspired probabilistic versions of paired semantics in [19] and Chapter 5). Our subsequent work presents how to automate pointer safety analysis and several value-related safety properties (including sign, non-zero, non-negativity) [17]. More recently, Simdiff [63] automates parts of the safety analysis from [16] using SMT solvers.

7.1.3 Search for Accuracy-Performance Tradeoffs

Profitably trading accuracy for performance is a well-known practice at an algorithmic level, where many algorithms have exposed knobs that control the level of approximations. In addition, accuracy-aware transformations introduce new (system-level) knobs. These knobs jointly induce an accuracy/performance tradeoff space. A configuration of a single program

typically consists of multiple knobs, one for each location where a transformation can be applied. By navigating this tradeoff space, search algorithms find configurations of knobs that result in profitable accuracy/performance tradeoffs.

Empirical Search. Researchers have presented various techniques for exploring accuracy/performance tradeoff space [5, 6, 9, 53, 75, 79, 95, 96, 100, 110, 113]. These techniques typically discretize the input space, by asking a developer to provide representative inputs, and discretize the configuration space, by trying the approximation with a fixed number of values for each knob (e.g., the configuration space of loop perforation in Chapter 2 has five distinct perforation rates). The search algorithms execute the transformed programs for various combinations of the knobs. These combinations are selected using various heuristic search strategies, such as exhaustive search (with optional pruning) [53, 95, 113], greedy search [79], genetic search [5, 6], or stochastic search [110].

Some of the search techniques are developer-guided, in that a developer manually exposes accuracy knobs, by selecting which locations in the program to transform) using program annotations [5, 6, 9, 46]. Other search techniques that automate the discovery of approximate subcomputations, check the safety of such candidate transformations using sensitivity profiling [75, 79, 80, 95, 96, 113], pattern identification [102], or type checking [104, 105].

The majority of thse techniques construct accuracy/performance models at compile-time. Alternatively, to improve the accuracy of the results, some techniques perform on-line recalibration, by occasionally running both the exact and approximate versions of the subcomputations [9, 46, 103]. Recently, Ding et al. [39] presented a variable-accuracy autotuning technique that, while searching for profitable tradeoffs at compile-time, finds configurations of approximate program that are appropriate for different input profiles.

Mathematical Optimization-Based Search. As an alternative to empirical search, this approach uses static analysis to construct accuracy constraints, which are then used as part of the formulation of an optimization problem. Then, existing numerical solvers can find the configuration of the approximate program that satisfies these constraints for all inputs of interest (while maximizing execution time or energy consumption objective).

We have previously used linear programming as a component of an approximation algorithm that finds an ε-optimal expected error/performance tradeoffs for map-fold computations automatically transformed using randomized program transformations, such as function substitution and sampling [125]. Chisel similarly uses mathematical programming to optimize for energy savings, while providing reliability and accuracy guarantees. In contrast to the approach from [125], which operates on an abstract model of computa-

tion, Chisel presents an analysis that operates on programming language statements, with randomness coming from approximate arithmetic operations and data.

In contrast to the majority of empirical search techniques, which cannot guarantee accuracy beyond representative inputs, optimization-based techniques can explore the tradeoff space to satisfy the constraints that are valid for all inputs of interest (that a developer describes as a part of the input specification). In addition, optimization based techniques can, in principle, operate on more complex configurations – e.g., a large number of possible transformations, as in Chisel, or a potentially infinite number of choices for selecting the probability of executing an approximate function version, as in [125]. However, optimization-based techniques are presently limited to computations with a specific structure.

7.2 Approximation at Intersection of Software and Hardware

Approximate Hardware Platforms. Researchers have proposed various designs of approximate hardware components. Approximate accelerators or cores expose coarse-grained operations that may occasionally fail or produce approximate results [66, 85, 121]. As a special case of coarse-grain approximate accelerators, researchers in academia and industry have presented accelerators that provide neural network abstractions [29, 43, 87, 117, 122].

Researchers have presented fine-grained approximate components, such as approximate ALUs and FPUs that produce bounded errors or fail with small probability (specified by the component designer) in return for a smaller size of the circuit and/or lower energy consumption [40, 42, 56, 61, 72, 89, 120, 124]. Researchers have also presented approximate main [67, 106] and cache memories [111] that can occasionally corrupt stored data. Many of these approximate hardware designs specify the frequency of failure of their components (e.g., an addition instruction may produce a wrong result with a small probability), and/or the magnitude of error (e.g., an addition instruction may produce a small bounded noise). Such components can, in principle, be represented using Chisel's approximate hardware model. In addition to intentionally designed approximate hardware, researchers have also investigated conditions under which commodity hardware experiences transient or intermittent errors caused by, e.g., power variation, temperature variation, or aging [123].

Overall, the diversity of approximate and unreliable hardware opens up new opportunities and challenges when optimizing programs at the compiler level. Chapter 4 presents Chisel's compiler support and analysis to optimize (with accuracy guarantees) programs running on a model of approximate hardware with approximate ALUs and memories (with approximate/unreliable components specified by the hardware designer).

Programming Models for Approximate Hardware. In previous work, we developed Rely [19], a language for expressing and analyzing computations that run on approximate hardware platforms. A developer can use Rely to manually select the operations and data that can execute unreliably using code annotations, such as "+." for approximate addition and "`int x in urel`" for a variable in approximate memory region. The developer also provides a reliability specification (that Chisel extends with energy and precision specifications). Rely's analysis verifies that a kernel with manually identified unreliable instructions and variables satisfies its reliability specification for all inputs.

In contrast to Chisel, Rely requires the developer to navigate the tradeoff between reliability and energy savings (because the developer is responsible for identifying the unreliable operations and data). Moreover, the developer must redo this identification every time the computation is ported to a new approximate hardware platform with different reliability and energy characteristics. Chisel leverages the reliability analysis approach from Rely to construct a combined reliability/accuracy specification, which enable Chisel to automatically optimize applications for different hardware specifications.

Flikker provides a set of C language extensions that enable a developer to specify data that can be stored in approximate memories [67]. EnerJ provides a type system that a developer can use to specify approximate data that can be stored in unreliable memory or computed using unreliable operations [105]. The EnerJ type system ensures the isolation of approximate computations and the compiler allocates data in approximate memories and assigns approximate instructions to execute on such data. Unlike Rely, Flikker, and EnerJ, which depend solely on the developer to identify exact and approximate operations and data, Chisel automates the selection of approximate operations and data while ensuring that the generated computation satisfies its reliability/accuracy specification.

Relax presents a combined hardware/software framework for detecting and recovering from hardware faults [32]. It allows a developer to provide recovery actions (such as reexecuting of a code block or discarding a part of the computation) to respond to to detected hardware faults. However, Relax requires specialized hardware organization to support recovery and it does not help a developer with analyzing how recovery blocks affect the accuracy, reliability, and safety of the computation.

ExpAX is a framework for expressing accuracy and reliability constraints for a subset of the Java language [91]. ExpAX uses a genetic programming optimization algorithm to search for approximations that minimize the energy consumption of the computation over a set of program traces. Chisel, in contrast, uses mathematical programming to guarantee

that the resulting program satisfies its reliability specification. In contrast, the genetic algorithm in ExpAX operates on representative inputs and provides no such guarantee.

Uncertain<T> is a type system for computing on approximate data from e.g., noisy sensors [13]. Uncertain<T> represents noise as probability distributions and dynamically computes probabilistic bounds on error as they propagate through the computation. It uses sampling and hypothesis testing to compute error distributions at different steps of the computation. In contrast to our analyses, it operates fully at run-time.

Topaz is a task-based language that allows the developer to specify tasks that execute on approximate hardware cores that may produce arbitrarily inaccurate results [2]. Topaz includes an outlier detector that identifies likely errors in the results of a program's tasks and a recovery strategy to efficiently reexecute or skip tasks. It operates with a coarser grain approximate hardware model and does not require a developer's accuracy specification.

7.3 Probabilistic Languages and Analyses

Representations of Uncertainty. Typical approaches for modeling uncertainty use intervals, random variables, or fuzzy sets to represent quantities that computations operate on. Interval analysis [83] represents uncertain quantities as intervals and defines basic arithmetic operations on such values (e.g., Section 5.3 describes one such analysis). It is often used to analyze the worst-case rounding error in numerical computations, ideally producing small error interval sizes. Additional knowledge about the inputs can make it possible to use probabilistic, fuzzy, or hybrid modeling of uncertainty in computations [50, 59]. This dissertation models uncertainty of inputs and program's execution via random variables (Chapter 3) and combination of random variables and intervals (Chapter 4).

Probabilistic Programming Languages. Researchers have presented languages for probabilistic modeling, in which programs work directly with probability distributions [36, 47, 60, 92, 93, 101]. These languages present constructs (statements, expressions, or built-in library calls) that introduce a random choice in the program execution. This dissertation formulates the probabilistic semantics of the Chisel's computations (Chapter 4) using implicitly random arithmetic operators (e.g., approximate addition) and operations on approximate data (e.g., reading from variable in approximate memory). We previously presented the semantics of these operators in the Rely language [19].

Probabilistic Analyses. Researchers have also presented numerous analyses to reason about probabilistic programs, including probabilistic axiomatic semantics, abstract interpretation, and model checking e.g., [37, 62, 81, 82, 84, 114]. More recently, researchers

have used symbolic execution to check probabilistic assertions that state that a program's property is correct with high probability. Filieri et al. [44] use symbolic execution and finite-state model checking to verify probabilistic assertions for programs that can operate on complex data structures. Sankaranarayanan et al. [108] present a symbolic analysis that checks for probabilistic assertions in programs with linear expressions and potentially unbounded loops. Claret et al. [28] use results of dataflow analysis to improve performance of probabilistic sampling procedures in Bayesian inference computations. The analysis in [21] checks for termination of probabilistic 'while'-loops by mapping the program semantics to martingales (a well-known class of stochastic processes). Sampson et al. [107] present a symbolic execution-based approach that translates a computation to a Bayesian network and use statistical sampling and/or algebraic transformations of random variables to approximately (with high confidence) verify probabilistic assertions.

Analysis of loop perforation in Chapter 3 quantitatively analyzes the application of loop perforation to a set of amenable computational patterns, which may appear in deterministic or probabilistic programs. It specifies probabilistic semantics at a pattern level instead of the statement level. In comparison with general probabilistic analyses, pattern-based analyses can, typically, provide more precise accuracy bounds, since patterns provide additional information about the nature of the analyzed computations that are instances of patterns. In addition, Chisel's analysis presents how to combine fully static reasoning about reliability and absolute error of an approximate computation, instead to reason about arbitrary probability distribution.

7.4 Analytic Properties of Programs

Continuity. Researchers have developed techniques to identify continuous or Lipschitz-continuous programs [24, 25, 68, 69, 94]. In addition to investigating sensitivity of computations to program transformations [25], additional domains in which these techniques have been applied include differential privacy [24, 25, 94] and analysis of robust functions for embedded systems, where noise comes from the inputs and environment [25, 68, 69]. For example, Chisel (Section 5.3) uses continuity property to calculate linear bounds on the propagation error for arithmetic operators, and used as a part of formulation of the Chisel's linear optimization problem.

Idempotence. Idempotence is the property that a computation that is executed multiple times on the same inputs produces the same result. Researchers have presented techniques that use this property of computations [33] to support inexpensive recovery from hardware faults and thus improve application's fault tolerance. In the context of approximate pro-

gram transformations, reexecution of idempotent computations can, in principle, be used to improve the reliability of approximate functions that use approximate arithmetic operations.

Smoothing. Smooth interpretation [23, 26] uses a gradient descent-based method to synthesize parameters of programs that control cyberphysical interactions. The analysis returns a set of parameters that minimize the difference between the expected and computed program's controlled values. It defines a smoothing semantics of computations, which produces a continuous approximation of the program's semantics (a smoothed program). Smooth interpretation then reduces synthesis of control parameters questions for this continuous program's semantics to a general (continuous, but potentially non-convex) mathematical optimization problems, that minimize the distance between the outputs the smoothed program and the desired outputs (or output distribution). This dissertation addresses a conceptually related problem – using numerical search to find the configurations of approximate programs. However, Chisel's approach is tailored to navigate the accuracy/performance tradeoff space to find programs that maximize performance subject to error magnitude and frequency for all inputs of interest.

7.5 Approximate Queries in Database Systems

Modern databases often enable users to define queries that operate on some subset of the records in a given table. Such queries come with no accuracy or performance guarantees. Researchers have explored multiple directions for supporting approximate queries with probabilistic guarantees. Approximate aggregate queries let a user specify a desired accuracy bound or execution time of a query [1, 54, 55]. The database then generates a sampling strategy that satisfies the specification [54, 55] or uses a cached sample, when applicable [1]. More recently, BlinkDB [3] presents an effective sampling procedure (based on aggregates of data in table columns) to provide approximate query results with confidence intervals. Online queries compute the exact answer for the entire data-set, but provide intermediate results and confidence bounds [51]. Probabilistic databases [30] operate on inherently uncertain data, and the accuracy bounds of all queries (including aggregation) depend on the statistics of uncertain data.

Similar to the techniques in approximate computing, these systems expose knobs that control accuracy and provide results with (probabilistic) error estimates. Such knobs can, in principle, be used as components in more complex approximate computations. But, unlike accuracy-aware transformations, these knobs are manually exposed by database developers and are applicable for a specific class of sampling queries.

8 Future Work

Program optimization using accuracy-aware transformations that I presented in this dissertation opens up a number of new research opportunities in approximate computing and software resilience. Going forward, this research can guide research on characterizing and exploiting approximation and uncertainty at every level of the computational stack – from algorithms to programming languages, compilers, runtime systems, and hardware.

Approximate programs can benefit from both algorithmic and system-level approximations at various levels of the computation stack (e.g., programming systems, runtime systems, operating systems, architecture), but only if programmers have tools to help them construct, manage, and reconfigure their approximate programs. The current research sets foundations for new general optimization systems that can reason about and optimize using a multitude of algorithmic-level and system-level approximation techniques.

Overall, I believe that this research direction can bring new powerful tools and techniques for better understanding approximation and uncertainty in computation. This includes applications that accuracy-aware compilers can automatically generate, but also the existing approximate computations, in which a developer is fully responsible for approximation, but uses tools to better understand the nature of his or her application and make more flexible and systematic approximation choices. Below, I describe several directions for extending the research presented in this dissertation.

Accuracy and Input Specification

Support for Deriving Accuracy Specifications. To systematically develop programs that use approximate software and hardware components, we need solid tool support to help with the process of deriving and debugging accuracy specification. This dissertation presents some of the first steps toward building an ecosystem of tools that help deliver approximate computations. Generalizing and improving the scalability of profiling, debugging, and testing techniques (such as sensitivity profiling) are essential for better understanding the properties of approximate computations and deriving appropriate formal accuracy requirement specifications.

Approximate Hardware Components. Chapter 4 presented a model of approximate hardware with arithmetic instructions and memory regions. However, it captures only some of the proposed approximate hardware architectures. As the field of approximate hardware is still in its early stages of development, we will likely require more expressive specifications of hardware-provided operations (including characteristics of their correct, approximate, and failed execution). It is likely that many such models will have nondeterministic and/or probabilistic components. The research presented in this dissertation, therefore, can be a good starting point for developing such more expressive models of approximate hardware.

Accuracy Analysis and Optimization

Analyses of an approximate computation's accuracy depend, in general, on the input specifications, the computation structure, and the accuracy specifications of the computation's components. To improve the capabilities of analysis of how uncertainty emerges and propagates through an arbitrary computation, future research needs to address a number of challenges, some of which are discussed below.

Analysis of Accuracy-Aware Transformations. This dissertation presents an optimization approach for generating approximate versions of approximate kernels that run on approximate hardware. Furthermore, our existing research on optimization of map-fold computations [125] (which operates on trees of "function nodes" representing map tasks and "reduction nodes" representing aggregation operators) presents a basis for the hierarchical analysis of computations and composition of accuracy specifications. As an immediate connection between these two techniques, we note that the optimization from Chapter 4 with arithmetic operations produces approximate versions of "function nodes" that can be used in optimization from [125]. As part of ongoing work and to support such applications, we are extending an approach from [125] with a program analysis that constructs constraints for the programs written using map and fold constructs [76].

Precision, Scalability, and Expressiveness of Accuracy Analysis. More expressive and diverse accuracy specifications require more powerful analyses that will ensure the correctness of developer specifications. In general, I anticipate that such analyses will present new tradeoffs between their scalability, precision, and expressiveness. To develop accuracy analyses that can verify or estimate with high confidence if an approximate computation satisfies its accuracy specification and facilitate automatic optimization, future analysis techniques can benefit from ideas and tools from numerical analysis, probabilistic inference, and verification of probabilistic systems.

Sensitivity of Accuracy Analysis and Optimization. Input specifications (e.g., intervals or distributions) and/or approximate hardware component specifications (e.g., failure probability) may change over time. An important future direction for adopting approximate computing includes accuracy analyses and optimization techniques that are robust to changes in the specifications (including input specifications, accuracy requirements and/or hardware platform).

As a related problem, the optimizations for even slightly different accuracy requirements or input specifications currently need to be performed from scratch. As an alternative, investigation of incremental optimization techniques that can (as a part of an application's runtime system) quickly recompute new optimization results, can enable the application to faster respond to the changes in its environment and inputs.

Program Repair and Resilience

Researchers have recently proposed techniques for automatically repairing software from general software errors. Some of these techniques, such as research on escaping from infinite loops using resiliency-oriented transformations [18, 58], generate code repairs that instruct the program to produce a partial or a different result as a consequence of steering the program execution past the error. For example, after escaping from an infinite loop, an application may continue executing from a program statement after the loop body, or from the return statement of one of the enclosing functions. In principle, an additional program transformation may also try to correct parts of the program state affected by the loop escape transformation.

The quality of the repaired program's output (and even the success of the continued execution) depends on the set of resiliency-oriented transformations used to repair the program. To navigate this tradeoff space induced by program repairs, future research can investigate how to adapt techniques used for the analysis of approximate programs to characterize the quality of the outputs that repaired programs produce and investigate techniques that can compose multiple resiliency-oriented transformations to maximize the quality of the repaired program's output.

9 Conclusion

Despite the central role that approximate computations play in many area of computer science, until recently the areas of program analysis and optimization have not used this potential to generate more flexible optimized programs. Only recently, researchers have started investigating compiler-level techniques for automatically generating approximate versions of programs that trade accuracy for improved performance and/or energy consumption.

This dissertation presents our investigation of the properties of accuracy-aware transformations and the foundation of rigorous analysis and optimization techniques that find acceptable tradeoffs between accuracy and performance/energy. Specifically, this dissertation identifies approximate kernels (subcomputations amenable to accuracy-aware transformations) and their structural and functional properties. It presents accuracy analysis techniques that analyze the frequency and magnitude of the noise that approximate transformations introduce. It presents how to automatically apply accuracy-aware transformations by formulating accuracy-aware program optimization as integer optimization problem. It presents experimental results that show that accuracy-aware transformations can discover profitable accuracy/performance tradeoffs that satisfy the developer's accuracy specifications.

In general, I see probabilistic accuracy analysis of approximate programs, with its ability to represent and automatically reason about uncertainty that arises in computation, and mathematical optimization, with its ability to find parameters that maximize or minimize an objective function while preserving a set of constraints, as a natural fit for many problems in approximate computing. Specifically, this dissertation presents how these general techniques can be used to optimize a resource consumption objective, such as energy or time, while providing acceptable execution, captured by accuracy constraints produced by program analysis.

A Transformed Chisel Kernels

This appendix presents the versions of approximate programs that Chisel successfully transformed (Chapter 6). We present the transformed kernels that obtained the maximum savings in the evaluation. Each approximate operation is denoted with an "." (e.g., "+."). Data in approximate memory is denoted with an annotation "`in unrel`". The original programs have the same reliability specification, but no data or instruction annotations.

```
extern float<1.0*R(v)> my_floor(float v);
extern float<1.0*R(v)> my_ceil(float v);
void SCALE_KERNEL(float factor,
       int(1)<1.0*R(src)> src in unrel, int sw, int sh,
       int(1)<0.995*R(factor,src, transformed,sw,sh,dw,dh,i,j,si,sj)>
                 transformed in unrel,
       int dw, int dh, int i, int j, float si, float sj)
{
  int previ = 0; int prevj = 0;
  int nexti = 0; int nextj = 0;
  float previf = 0; float prevjf = 0;
  float nextif = 0; float nextjf = 0;
  float lr0 = 0; float lr1 = 0; float lr2 = 0;
  float ll0 = 0; float ll1 = 0; float ll2 = 0;
  float ul0 = 0; float ul1 = 0; float ul2 = 0;
  float ur0 = 0; float ur1 = 0; float ur2 = 0;
  previ = (int)my_floor(si); prevj = (int) my_floor(sj);
  nexti = (int) my_ceil(si); nextj = (int) my_ceil(sj);
  if (sh <= previ) { previ = sh - 1; }
  if (sw <= prevj) { prevj = sw - 1; }
  if (sh <= nexti) { nexti = sh - 1; }
  if (sw <= nextj) { nextj = sw - 1; }
  if (previ == nexti) {
        if (0 == previ) {
           if  (sw - 1 == nexti) { }
           else { nexti = nexti + 1; }
        } else { previ = previ - 1; }
  }
  // continued on the next page ...
```

Figure A.1: Scale Kernel Generated for Configuration M/M/M (part 1)

```
// ... continued from the previous page
if (prevj == nextj) {
     if (0 == prevj) {
         if (sh - 1 == nextj) {}
         else { nextj = nextj + 1; }
     } else { prevj = prevj - 1; }
}
lr0 = src[(3*(previ*sw+prevj))];
lr1 = src[(3*(previ*sw+prevj)+1)];
lr2 = src[(3*(previ*sw+prevj)+2)];
ll0 = src[(3*(previ*sw+nextj))];
ll1 = src[(3*(previ*sw+nextj)+1)];
ll2 = src[(3*(previ*sw+nextj)+2)];
ur0 = src[(3*(nexti*sw+prevj))];
ur1 = src[(3*(nexti*sw+prevj)+1)];
ur2 = src[(3*(nexti*sw+prevj)+2)];
ul0 = src[(3*(nexti*sw+nextj))];
ul1 = src[(3*(nexti*sw+nextj)+1)];
ul2 = src[(3*(nexti*sw+nextj)+2)];
previf = (float) previ; prevjf = (float) prevj;
nextif = (float) nexti; nextjf = (float) nextj;
transformed[(3*(i*dw+j))]= (int)
     (1.0/((nextif-previf)*(nextjf-.prevjf))
         *(lr0*(nextjf-sj)*(nextif-si)+.ur0*.(nextjf-sj)
         *(si-.previf)+ll0*(sj-prevjf)*.(nextif-si)+ul0
         *.(sj-prevjf)*.(si-previf)));
transformed[(3*(i*dw+j)+1)]= (int)
     (1.0/((nextif-previf)*.(nextjf-prevjf))
         *.(lr1*(nextjf-sj)*.(nextif-si)+.ur1*(nextjf-sj)
         *(si-previf)+ll1*(sj-prevjf)*.(nextif-si)+.ul1
         *.(sj-.prevjf)*(si-.previf)));
transformed[(3*(i*dw+j)+2)]= (int)
     (1.0/((nextif-previf)*.(nextjf-prevjf))
         *.(lr2*.(nextjf-.sj)*.(nextif-si)+.ur2*(nextjf-.sj)
         *(si-previf)+ll2*.(sj-prevjf)*.(nextif-si)+.ul2
         *(sj-.prevjf)*(si-previf)));
}
```

Figure A.2: Scale Kernel Generated for Configuration M/M/M (part 2)

```
void DCT_KERNEL(int(1)<1.0*R(src)> src, int sw, int sh,
      int(1)<0.99992*R(src,sw,sh,dest,dw,dh,i,j,curri,currj)>
            dest in unrel,
      int dw, int dh, int i, int j, int curri, int currj)
{
    int di; int dj;
    int src_pix_r; int src_pix_g; int src_pix_b;
    int dest_pix_r; int dest_pix_g; int dest_pix_b;
    float currif; float currjf; float dif; float djf;
    int dred; int dgreen; int dblue;
    dred = 0; dgreen = 0; dblue = 0;
    di = 0;
    while(i + di < dh) : 8 {
      dj = 0;
      while(j + dj < dw) : 8 {
        dest_pix_r = dred; dest_pix_g = dgreen; dest_pix_b = dblue;
        src_pix_r = src[(3*(((i+di)*(sw)+(j+dj))))];
        src_pix_g = src[(3*(((i+di)*(sw)+(j+dj)))+1)];
        src_pix_b = src[(3*(((i+di)*(sw)+(j+dj)))+2)];
        currif = curri; currjf = currj;
        dif = di; djf = dj;
        dred =(int) (dest_pix_r + (src_pix_r-128)
                *4*my_cos(3.14159/8*(djf + 0.5)*currjf)
                *my_cos(3.14159/8*(dif + 0.5)*currif));
        dgreen =(int) (dest_pix_g + (src_pix_g-128)
                *4*my_cos(3.14159/8*(djf + 0.5)*currjf)
                *my_cos(3.14159/8*(dif + 0.5)*currif));
        dblue =(int) (dest_pix_b + (src_pix_b-128)
                *4*my_cos(3.14159/8*(djf + 0.5)*currjf)
                *my_cos(3.14159/8*(dif + 0.5)*currif));
        dj = dj + 1;
      }
      di = di + 1;
    }
    dest[(3*(((i+curri)*(dw)+(j+currj))))] = dred;
    dest[(3*(((i+curri)*(dw)+(j+currj)))+1)] = dgreen;
    dest[(3*(((i+curri)*(dw)+(j+currj)))+2)] = dblue;
}
```

Figure A.3: DCT Kernel Generated for Configuration M/m/M

```
extern float<1.0*R(v)> my_cos(float v);
extern float<1.0*R(v)> my_sin(float v);
void IDCT_KERNEL(
       int(1) src in unrel, int sw, int sh,
       int(1)<0.992*R(src,sw,sh,dest,dw,dh,i,j,curri,currj)>
              dest in unrel, int dw, int dh, int i, int j,
              int curri, int currj )
{
    int di; int dj; int currk=0;
    float dr; float dg; float db;
    int src_pix_r; int src_pix_g; int src_pix_b;
    float w1; float w2;
    float currif; float currjf; float dif; float djf;
    di = 0; dr = 0; dg = 0; db = 0;
    while(di +. i < dh) :8 {
       dj = 0;
       while(dj + j < dw) : 8 {
         if(di == 0){ w1 = 0.5; } else { w1 = 1.0; }
         if(dj == 0){ w2 = 0.5; } else{ w2 = 1.0; }
         currif = curri; currjf = currj; dif = di; djf = dj;
         src_pix_r = src[(3*(((i+di)*(sw)+(j+dj))))];
         src_pix_g = src[(3*(((i+di)*(sw)+(j+dj)))+1)];
         src_pix_b = src[(3*(((i+di)*(sw)+(j+dj)))+2)];
         dr = (dr + (src_pix_r)*w1*w2
                  *my_cos(3.14159/8*(currjf + 0.5)*djf)
                  *my_cos(3.14159/8*(currif + 0.5)*dif));
         dg = (dg + (src_pix_g)*w1*w2
                  *my_cos(3.14159/8*(currjf + 0.5)*djf)
                  *my_cos(3.14159/8*(currif + 0.5)*dif));
         db = (db + (src_pix_b)*w1*w2
                  *my_cos(3.14159/8*(currjf + 0.5)*djf)
                  *my_cos(3.14159/8*(currif + 0.5)*dif));
         dj = dj + 1;
       }
       di = di +. 1;
  }
  dr = dr/.(8*8)+128; dg = dg/.(8*8)+.128; db = db/.(8*8)+.128;
  if(dr <. 0){ dr = 0; } if(dr >. 255){ dr = 255; }
  if(dg <. 0){ dg = 0; } if(dg >. 255){ dg = 255; }
  if(db <. 0){ db = 0; } if(db > 255) { db = 255;  }
  dest[(3*(((i+curri)*(dw)+(j+currj))))] = (int) dr;
  dest[(3*(((i+curri)*(dw)+(j+currj)))+1)] = (int) dg;
  dest[(3*(((i+curri)*(dw)+(j+currj)))+2)] = (int) db;
}
```

Figure A.4: IDCT Kernel Generated for Configuration M/m/m

```
extern float<1.0*R(f)> rel_fabs(float f);
extern float<1.0*R(f)> rel_exp(float f);
extern float<1.0*R(f)> rel_sqrt(float f);
extern float<1.0*R(f)> rel_log(float f);
float<0.999*R(sptprices,strikes,rates,volatilities,times,otypes,timet,index)>
    BLACKSCHOLES_KERNEL(
           float(1)<1.0*R(sptprices)> sptprices in unrel,
           float(1)<1.0*R(strikes)> strikes in unrel,
           float(1)<1.0*R(rates)> rates in unrel,
           float(1)<1.0*R(volatilities)> volatilities in unrel,
           float(1)<1.0*R(times)> times,
           int(1)<1.0*R(otypes)> otypes in unrel,
           float timet, int index)
{
    float OutputX; float xNPrimeofX; float xK2; float xK2_2;
    float xK2_3; float xK2_4; float xK2_5; float xLocal;
    float OptionPrice; float xDen; float d1; float d2;
    float FutureValueX; float NofXd1; float NofXd2;
    float NegNofXd1; float NegNofXd2;
    float sptprice; float strike; float rate;
    float xVolatility; float time; int otype; bool sign;
    sptprice = sptprices[index]; strike = strikes[index];
    rate = rates[index]; xVolatility = volatilities[index];
    time = times[index]; otype = otypes[index];
    d1 = (rate +. 0.5 * xVolatility * xVolatility) * time
           + rel_log( sptprice / strike );
    xDen = xVolatility * rel_sqrt(time);
    d1 = d1 / xDen; d2 = d1 - xDen;
  // continued on the next page ...
```

Figure A.5: Blackscholes Kernel Generated for Configuration M/m/m (part 1)

```
    // ... continued from the previous page
    if (d1 < 0.0) {d1 = 0.0-d1; sign = true;
    } else { sign = false; }
    xNPrimeofX = rel_exp(0.0-0.5 * d1 * d1) * 0.398942;
    xK2 = 1.0 / (1.0 + 0.2316419 * d1);
    xK2_2 = xK2 * xK2; xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2; xK2_5 = xK2_4 * xK2;
    xLocal = xK2_2 * (0.0-0.356563782) + xK2_3 * 1.781477937
               + xK2_4 * (0.0-1.821255978) + xK2_5 * 1.330274429
               + xK2 * 0.319381530;
    NofXd1 = 1.0 -. xLocal * xNPrimeofX;
    if (sign) { NofXd1 = 1.0 - NofXd1; }
    if (d2 < 0.0) { d2 = 0.0-d2; sign = true; }
    else { sign = false; }
    xNPrimeofX = rel_exp(0.0 -0.5 * d2 *. d2) * 0.3989422;
    xK2 = 1.0 / (1.0 + 0.2316419 * d2);
    xK2_2 = xK2 * xK2; xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2; xK2_5 = xK2_4 * xK2;
    xLocal = xK2_2 * (0.0-0.356563782) +. xK2_3 * 1.781477937
               + xK2_4 * (0.0-1.821255978) + xK2_5 * 1.330274429
               + xK2 * 0.319381530;
    NofXd2 = 1.0 - xLocal * xNPrimeofX;
    if (sign) { NofXd2 = 1.0 - NofXd2; }
    FutureValueX = strike * ( rel_exp( (0.0-.rate)*(time) ) );
    if (otype == 0) {
        OptionPrice = (sptprice * NofXd1) - (FutureValueX * NofXd2);
    } else {
        OptionPrice = (FutureValueX *(1.0 - NofXd2) ) - (sptprice * (1.0 - NofXd1));
    }
    return OptionPrice;
}
```

Figure A.6: Blackscholes Kernel Generated for Configuration M/m/m (part 2)

```
float<0.995*R(G,i,j,N,omega)> SOR_KERNEL(
        float(1)<1.0*R(G)> G in unrel, int i, int j, int N, float omega)
{
   float omega_over_four;
   float one_minus_omega;
   float up;
   float down;
   float left;
   float right;
   float center;
   omega_over_four = omega *. 0.25;
   one_minus_omega = 1.0 -. omega;
   up=G[((i-.1)*.N +. j)];
   down=G[((i+.1)*N + j)];
   left=G[((i)*.N +. j-.1)];
   right=G[((i)*.N + j+.1)];
   center=G[((i)*N +. j)];
   return omega_over_four *. (up +. down +. left +. right) +.
        one_minus_omega *. center;
}
```

Figure A.7: Sor Kernel Generated for Configuration M/M/M

Bibliography

[1] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. SIGMOD, 1999.

[2] S. Achour and M. Rinard. Energy efficient approximate computation with topaz. OOPSLA, 2015.

[3] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. Eurosys, 2013.

[4] E. Amigo, J. Gonzalo, and J. Artiles. A comparison of extrinsic clustering evaluation metrics based on formal constraints. Information Retrieval Journal. Springer Netherlands, July 2008.

[5] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. PACT, 2014.

[6] J. Ansel, Y. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. CGO, 2011.

[7] B. Arnold, N. Balakrishnan, and H. Nagaraja. *A first course in order statistics.* Society for Industrial Mathematics, 2008.

[8] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[9] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[10] T. Bao, Y. Zheng, and X. Zhang. White box sampling in uncertain data processing enabled by program analysis. OOPSLA, 2012.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. PACT, 2008.

[12] J. Birge. *Optimization Methods in Dynamic Portfolio Management (Chapter 20).* Elsevier, 2007.

[13] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain¡T¿: A first-order type for uncertain data. ASPLOS, 2014.

[14] A. C. Bovik. *Handbook of image and video processing.* Academic press, 2010.

[15] M. Carbin. *Reasoning about Approximate Computing.* PhD thesis, EECS Department, MIT, Feb 2015.

[16] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[17] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Verified integrity properties for safe approximate program transformations. PEPM, 2013.

[18] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. ECOOP, 2011.

[19] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA, 2013.

[20] M. Carbin and M. Rinard. Automatically identifying critical input regions and code in applications. ISSTA, 2010.

[21] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. CAV, 2013.

[22] S. Chakradhar, A. Raghunathan, and J. Meng. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. IPDPS, 2009.

[23] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. volume 49 of *ACM SIGPLAN Notices*, pages 207–220. ACM, 2014.

[24] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. POPL, 2010.

[25] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. FSE, 2011.

[26] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. PLDI, 2010.

[27] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. Proceedings of the 50th Annual Design Automation Conference, page 113. ACM, 2013.

[28] G. Claret, S. Rajamani, A. Nori, A. Gordon, and J. Borgström. Bayesian inference using data flow analysis. FSE, 2013.

[29] Introducing Qualcomm Zeroth Processors: Brain-Inspired Computing. `https://www.qualcomm.com/news/onq/2013/10/10/introducing-qualcomm-zeroth-processors-brain-inspired-computing`.

[30] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Journal of CACM*, 2009.

[31] E. Darulova and V. Kuncak. Sound compilation of reals. POPL, 2014.

[32] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. ISCA, 2010.

[33] M. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. PLDI, 2012.

[34] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[35] J. Deutscher and I. Reid. Articulated body motion capture by stochastic search. *International Journal of Computer Vision*, 2005.

[36] A. Di Pierro, C. Hankin, and H. Wiklicky. Probabilistic λ-calculus and quantitative program analysis. *Journal of Logic and Computation*, 2005.

[37] A. Di Pierro and H. Wiklicky. Concurrent constraint programming: Towards probabilistic abstract interpretation. PPDP, 2000.

[38] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8), August 1975.

[39] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, June 2015.

[40] P. Düben, J. Joven, A. Lingamneni, H. McNamara, G. De Micheli, K. Palem, and T. Palmer. On the use of inexact, pruned hardware in atmospheric modelling. *Philosophical Transactions of the Royal Society*, 372, 2014.

[41] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. ISCA, 2011.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. ASPLOS, 2012.

[43] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. MICRO, 2012.

[44] A. Filieri, C. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. ICSE, 2013.

[45] A. Gaffar, O. Mencer, W. Luk, P. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. FPT, 2002.

[46] I. Goiri, R. Bianchini, S. Nagarakatte, and T. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. ASPLOS, 2015.

[47] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. UAI, 2008.

[48] Gurobi. http://www.gurobi.com/ .

[49] M. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Interprocedural analysis for parallelization. *Languages and Compilers for Parallel Computing*, 1996.

[50] J. Halpern. *Reasoning about uncertainty*, volume 21. MIT Press Cambridge, 2003.

[51] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. SIGMOD, 1997.

[52] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, MIT, September 2009.

[53] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[54] W. Hou, G. Ozsoyoglu, and B. Taneja. Processing aggregate relational queries with hard time constraints. SIGMOD, 1989.

[55] Y. Hu, S. Sundara, and J. Srinivasan. Supporting time-constrained sql queries in oracle. VLDB, 2007.

[56] H. Kaul, M. Anders, S. Mathew, S. Hsu, A Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. ISSCC, 2012.

[57] K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufman, 2002.

[58] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. OOPSLA, 2012.

[59] G.J. Klir. *Uncertainty and information.* John Wiley & Sons, 2006.

[60] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 1981.

[61] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. VLSI Design, 2011.

[62] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker. *Computer Performance Evaluation: Modelling Techniques and Tools*, 2002.

[63] S. Lahiri, A. Haran, S. He, and Z. Rakamaric. Automated differential program verification for approximate computing. Technical report, May 2015.

[64] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO, 2004.

[65] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. CASES, 2006.

[66] L. Leem, H. Cho, J. Bau, Q. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. DATE, 2010.

[67] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[68] R. Majumdar and I. Saha. Symbolic robustness analysis. RTSS, 2009.

[69] R. Majumdar, I. Saha, and Z. Wang. Systematic testing for control applications. MEMOCODE, 2010.

[70] Xiph.org Video Test Media. http://media.xiph.org/video/derf.

[71] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna. Exploiting the forgiving nature of applications for scalable parallel execution. IPDPS, 2010.

[72] J. Miao, A. Gerstlauer, and M. Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. ICCAD), 2013.

[73] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. OOPSLA, 2014.

[74] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Reliability-aware optimization of approximate computational kernels with rely. Technical Report MIT-CSAIL-TR-2014-001, MIT, 2014.

[75] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS Special Issue on Probabilistic Embedded Computing*, 2013.

[76] S. Misailovic and M. Rinard. Synthesis of randomized accuracy-aware map-fold programs. Technical Report MIT-CSAIL-TR-2013-031, MIT, 2013.

[77] S. Misailovic, D. Roy, and M. Rinard. Probabilistic and Statistical Analysis of Perforated Patterns. Technical Report MIT-CSAIL-TR-2011-003, MIT, 2011.

[78] S. Misailovic, D. Roy, and M. Rinard. Probabilistically accurate program transformations. SAS, 2011.

[79] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[80] S. Misailovic, S. Sidiroglou, and M. Rinard. Dancing with uncertainty. RACES, 2012.

[81] D. Monniaux. Abstract interpretation of probabilistic semantics. SAS, 2000.

[82] D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. POPL, 2001.

[83] R.E. Moore. *Interval analysis.* Prentice-Hall, 1966.

[84] C. Morgan and A. McIver. pGCL: formal reasoning for random algorithms. *South African Computer Journal*, 22, 1999.

[85] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. DATE, 2010.

[86] N. Nethercote and J. Seward. Valgrind A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[87] New IBM SyNAPSE Chip Could Open Era of Vast Neural Networks. `http://www-03.ibm.com/press/us/en/pressrelease/44529.wss`.

[88] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. FPL, 2007.

[89] K. Palem. Energy aware computing through probabilistic switching: A study of limits. *IEEE Transactions on Computers*, 2005.

[90] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. Flexjava: Language support for safe and modular approximate programming. FSE, 2015.

[91] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-14-05, Georgia Institute of Technology, 2014.

[92] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. POPL, 2005.

[93] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. POPL, 2002.

[94] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. ICFP, 2010.

[95] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[96] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[97] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. OSDI, 2004.

[98] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. ASPLOS, 2015.

[99] P. Roy, R. Ray, C. Wang, and W. Wong. Asac: automatic sensitivity analysis for approximate computing. LCTES, 2014.

[100] C. Rubio-González, C. Nguyen, H. Nguyen, J. Demmel, W. Kahan, K. Sen, D. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. SC, 2013.

[101] N. Saheb-Djahromi. Probabilistic LCF. MFCS, 1978.

[102] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. ASPLOS, 2014.

[103] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. MICRO, 2013.

[104] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. Technical report, 2015.

[105] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. PLDI, 2011.

[106] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. MICRO, 2013.

[107] A. Sampson, P. Panchekha, T. Mytkowicz, K. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. PLDI, 2014.

[108] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. PLDI, 2013.

[109] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. LCTES, 2002.

[110] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. PLDI, 2014.

[111] M. Shoushtari, A. Banaiyan, and N. Dutt. Relaxing manufacturing guard-bands in memories for energy savings. Technical Report CECS TR 10-04, UCI, 2014.

[112] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis. Assure: automatic software self-healing using rescue points. ASPLOS, 2009.

[113] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE, 2011.

[114] M. Smith. Probabilistic abstract interpretation of imperative programs using truncated normal distributions. *Electronic Notes in Theoretical Computer Science*, 2008.

[115] M.D. Springer. *The algebra of random variables.* John Wiley & Sons, 1979.

[116] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. From program verification to program synthesis. POPL, 2010.

[117] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. ISCA, 2014.

[118] Parsec Benchmark Suite. http://parsec.cs.princeton.edu/.

[119] SciMark2 Benchmark Suite. math.nist.gov/scimark2/.

[120] J. Tong, D. Nagle, and R. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integrated Systems*, 2000.

[121] S. Venkataramani, V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. MICRO, 2013.

[122] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: energy-efficient neuromorphic systems using approximate computing. ISLPED, 2014.

[123] L. Wanner, L. Lai, A. Rahimi, M. Gottscho, P. Mercati, C. Huang, F. Sala, Y. Agarwal, L. Dolecek, N. Dutt, et al. Nsf expedition on variability-aware software: Recent results and contributions. *Information Technology*, 57(3):181–198, 2015.

[124] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu. On reconfiguration-oriented approximate adder design and its application. ICCAD. IEEE Press, 2013.

[125] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.