Scalable and Broad Hardware Acceleration Through Practical Speculative Parallelism

by

Maleen Abeydeera

B. S. University of Moratuwa, Sri Lanka (2014)
M. S. Massachusetts Institute of Technology (2017)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author .....
Department of Electrical Engineering and Computer Science
June 29, 2021

Certified by .....
Daniel Sanchez
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Scalable and Broad Hardware Acceleration Through Practical Speculative Parallelism

by

Maleen Abeydeera

Submitted to the Department of Electrical Engineering and Computer Science
on June 29, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

With the slowing down of Moore's Law, silicon fabrication technology is not yielding the performance improvements it once did. Hardware accelerators, which tailor their architecture to a specific application or domain, have emerged as an attractive approach to improve performance. Unfortunately, current accelerators have been limited to domains such as deep learning, where parallelism is easy to exploit. Many applications do not have such easy-to-extract parallelism and have remained off-limits to accelerators.

This thesis presents techniques to build accelerators for applications with speculative parallelism. These applications consist of atomic tasks, sometimes with order constraints, and need speculative execution to extract parallelism. In speculative execution, tasks are executed in parallel assuming they are independent. A runtime system monitors their execution to see if they are. If a task produces a conflict during execution, i.e., if it may violate a data dependence, then it is aborted and re-executed.

This thesis proposes Chronos, a framework-based approach for building accelerators that use speculation to extract parallelism. Under Chronos, accelerator designers express the algorithm as a set of ordered tasks, and then design processing elements (PEs) to execute each of these tasks. The framework provides reusable components for task management and speculative execution, saving most of the developer effort in creating accelerators for new applications.

Prior general-purpose architectures have leveraged already existing techniques, like cache-coherence protocols, for conflict detection, but implementing coherence would add complexity, latency and significant on-chip storage requirement, making these techniques expensive on accelerators.

To tackle this challenge, we first propose a new execution model, Spatially Located Ordered Tasks (SLOT), that uses order as the only synchronization mechanism and limits task accesses to a single read-write object. We then use SLOT to implement the Chronos framework. This implementation avoids the need for cache coherence and makes speculative execution cheap and distributed. This reduces overheads and improves performance by up to 2× over prior conflict detection techniques.

While SLOT achieves excellent performance on many algorithms, it is sometimes

desirable to allow a single task to access multiple objects. Thus, we extend Chronos to support the more general Swarm execution model, which allows this and is also easier to program. This Chronos-Swarm implementation improves performance when Swarm's features are needed, but it hurts performance when they are not, as the Swarm execution model requires more expensive conflict checks on each memory access. To bridge this gap, we introduce a hybrid SLOT/Swarm execution model that combines the generality and ease-of-programming of Swarm with the performance of SLOT.

We develop FPGA implementations of Chronos and use them to build accelerators for several challenging applications. When run on cloud FPGA instances, these accelerators outperform state-of-the-art software versions running on a higher-priced multicore instance by 3.5× to 15.3×.

Thesis Supervisor: Daniel Sanchez
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank the many people who helped me in this Ph.D. journey.

First, a big thank you goes to my advisor, Professor Daniel Sanchez. I started from a different background than computer architecture, and Daniel quickly helped get me up to speed in this area. He helped me find the perfect project that matched with my interest in building large-scale FPGA systems and helped present it to the wider architecture community. Daniel was always reachable regardless of how busy he was, and when things did not always go according to plan, he was always available to find solutions.

Next, I would like to thank my thesis committee members, Professor Joel Emer and Professor Arvind. Joel has taught me how to think clearly from first principles, and how using the right terminology always helps to present ideas clearly. Arvind also provided valuable feedback that helped improve this thesis when I first presented it to him in my qualifying exam.

Over the six years I have had spent at MIT, I had the priviledge of working alongside Mark Jeffrey, Suvinay Subramanian, Victor Ying, Hyun Ryong (Ryan) Lee, and Anurag Mukkara. I have learned a lot from each one of them. I am also grateful for the other members of the Sanchez group, Nathan Beckmann, Nosayba El-Sayed, Axel Feldmann, Harshad Kasture, Quan Nguyen, Nikola Samardzic, Po-An Tsai, Yifan Yang, and Guowei Zhang. Apart from providing insightful technical feedback on our work, they made the experience of working in a group environment fun and relaxed. I will forever treasure the memories made with each one of them.

I am also thanksful to the wonderful group of Sri Lankan friends I have had in the Boston area. You guys have made sure that I always had something fun to look forward to in the weekends.

Last but not least, none of this would be possible without my lovely wife, Sandamali Devadithya. She changed schools midway through her own Ph.D. journey to be with me. She stood beside me at all times, did not let me give up, and gave me confidence that I could finish the journey that I started. I realize how lucky I am to have you in my life, and I am looking forward to many years of adventures together.

I am grateful for financial support from NSF grants CAREER-1452994 and SHF-1814969, NSF/SRC grant E2CDA-1640012, and by a Sony research grant.

Contents

CHAPTER 1

Introduction

Over the past several decades, computer systems have relied on Moore's Law [72], which predicted that the number of transistors on a chip will double every two years, to deliver exponentially increasing performance. However, this trend is expected to slow down as transistors are nearing their physical limits [60]. In addition, since the mid-2000s, power density has become the key limitation of processor performance [32], and has limited the performance benefits of Moore's Law.

To keep increasing performance, computer systems must use limited resources more efficiently. A promising approach to achieve this is to use hardware accelerators that are tailored to a specification application or domain. Some accelerators are specialized to a single application, such as a video decoder for a single protocol [1,3], while others allow for some programmability that allows targeting multiple applications on a particular domain, such as deep learning [23, 57] or graphics [15, 96]. Most modern devices consist of a heterogeneous mix of such accelerators. For example, a cell phone contains application-specific accelerators like baseband processors, and programmable ones like GPUs [21, 24].

Accelerators rely on exploiting large amounts of parallelism, since the only way of utilizing the billions of transistors present in a modern chip is to perform many activities in parallel. Unfortunately, prior accelerators have focused on domains where parallelism is easy to exploit, like deep learning [23, 57], video processing [1, 3] and graphics [15, 96]. However, many applications do not have such easy-to-extract parallelism, and have remained off-limits to accelerators.

To understand why, we now review how parallelizing an application works in general. Starting from a sequential algorithm, the work in this algorithm must be divided into units of work that may be run in parallel. In this thesis, we call these units of work *tasks*. These tasks need to mapped to multiple parallel execution units, like processor

cores or specialized processing elements. However, these tasks will have data dependences since they are operating in the same data. Therefore, we need some means of synchronization to ensure that parallel tasks preserve these data dependences.

Prior accelerators focus on applications where tasks and their dependences are known in advance. They rely on conventional parallelization techniques, such as data-parallel or pipelined execution to exploit this parallelism cheaply. These kind of problems are common in deep learning, genomics, and signal processing.

By contrast, in this thesis we focus on applications where tasks or their dependences are unknown in advance. These kind of problems are common in graph analytics, simulation, and databases. For these kinds of applications, parallelism is harder to extract. A possible approach is to examine the set of available tasks, analyze their dependences, and then use that information to have them synchronize appropriately. However, for some applications, like discrete-event simulation, the work of finding which tasks are independent overwhelms the application work (Section 2.3.1).

For these applications, it is more beneficial to speculate. Speculation is a general technique to deal with data dependences, where the system makes a guess when it does not have advance knowledge of a dependence or value. It then continues execution assuming that this guess is correct. If it is correct, then speculation has succeeded. If not, the speculation has failed and the system must abort some of the ongoing work and re-execute it to preserve correct behavior. In our case, we speculate that the tasks are independent, and monitor their execution to see if they are. If a task produces a *conflict* during execution, i.e., if it may violate a data dependence, it is aborted and re-executed.

1.1 Challenges

There is a rich body of work on applying speculative execution to general-purpose parallel systems like multicore chips. However, there has been limited prior work in applying speculation to accelerators. This causes several challenges.

No reusable foundation for building accelerators that perform task-level speculation

Ideally, we would like to have a design framework that enables us to design accelerators quickly. A framework provides a structure to build accelerators where some part of the design is application-specific, but the overall structure of the accelerator follows a defined template. Such a framework amortizes the design effort of creating a new accelerator; it enables reuse of the hardware template as well as any tools like compilers that could be used across different instances of this template. Unfortunately, such a framework is not available for applications with speculative parallelism.

There are several examples of such successful frameworks for other kinds of applications. For example, consider a systolic array [59], which consists of a grid of processing elements (PEs) that perform operations and communicate values to nearby PEs in a synchronous fashion. Systolic array frameworks, such as LSSD [75] or Gemmini [38], enable accelerator development by following this template, leaving the design of the PE to the accelerator designer and providing support for everything else, including the fabric that connects the PEs and clear interfaces for design of PEs. Some of the frameworks also provide support for compilation, where a compiler extracts a dataflow graph from a higher-level description, and then maps the nodes in this graph to PEs.

Such a framework-based approach may trade performance in comparison to developing an accelerator from scratch, but it often facilitates faster accelerator development by easing design complexity, and can achieve performance close to that of an ASIC [75].

Some frameworks like ParallelXL [22] and TAPAS [69] have targeted extracting non-speculative parallelism from applications with dynamic tasks or unknown data dependences. These frameworks have proposed different execution models (e.g., continuation-passing style, fork-join parallelism) for expressing the program. They also provide necessary compilers for generating PEs and the hardware structures needed for task scheduling.

But these frameworks do not support speculative parallelism. In fact, the need for a framework is more apparent with speculation since performing task-level speculative execution is a complex but largely application-independent operation. All applications that perform speculation need a way to track read/write sets of tasks to identify conflicts, and to resolve each conflict by aborting one task and rolling back its effects. In addition, if tasks have strict ordering requirements, there needs to be a mechanism of storing and dispatching these tasks in priority order.

Prior general-purpose speculation mechanisms are too expensive for accelerators

To make speculation efficient, prior work in multicore systems has proposed hardware support for speculation, including Thread-Level Speculation [37,84,89,91], Hardware Transactional Memory [4,13,19,36,44,47,50,73], and systems that support speculative ordered tasks [54,55,93,99]. However, all of these prior general-purpose systems *reuse existing mechanisms* to implement speculative execution. On a memory access, all these systems need to locate, across the whole system, other conflicting tasks that accessed the same line. This is often done by adapting the cache coherence protocol.

Coherence-based conflict detection works by leveraging invalidation and downgrade messages to detect conflicts. Each task runs in a single core. The core acquires coherence permissions for each accessed cache line as usual, but keeps permissions for these lines throughout the execution of the task. The core then detects every possible conflict because the coherence protocol notifies it of reads and writes from other cores through invalidations and downgrades.

While relying on coherence is reasonable for multicores, it can be expensive for

accelerators. Accelerators in general and reconfigurable hardware in particular do not have a coherent cache hierarchy that supports invalidation-based conflict detection [87]. Implementing coherence would add complexity, latency, and significant on-chip SRAM to implement a directory that tracks sharers, and to implement structures that track read/write sets as we will see in Chapter 4.

Additionally, prior general-purpose techniques relied on non-standard hardware primitives. For instance, Swarm's proposed implementation [54] uses TCAMs for implementing priority queues, and implements efficient access to many Bloom filters by using SRAM banks with perpendicular ports to support row-wise access for reads and column-wise access for writes. Such structures are rarely available on standard cell libraries and definitely not available on FPGAs.

1.2 Contributions

To address these challenges, this thesis makes the following contributions.

1.2.1 Chronos framework for accelerators

This thesis presents Chronos, a framework-based approach for building accelerators that use speculation to extract parallelism.

Under Chronos, accelerator designers express the algorithm as a set of ordered tasks, where order is specified as a programmer-defined timestamp associated with each task. The designers then build processing elements (PEs) to execute each of these tasks. These PEs have a defined interface to convey newly created tasks to hardware, dequeue new tasks for execution, and perform memory accesses.

Chronos provides a reusable framework to run these tasks speculatively and out-of-order. The framework provides the task management logic, implements a memory system that allows tasks to perform arbitrary accesses, and the logic necessary to recover from misspeculation. This framework is highly customizable to the specific needs and semantics of the application. In practice, this framework-based approach ensures that around 95% of the code is reused across applications.

Each Chronos instance consists of spatially distributed tiles. Each tile has multiple PEs that execute tasks, and a local cache. Each tile also implements hardware to queue tasks, dispatch them to PEs, track their speculative state, and abort or commit them in timestamp order. Accelerators can then specialize this template by defining their own PEs. Chronos provides a carefully designed interface that allows many different PE implementation styles (e.g., simple Finite State Machines, deeply-pipelined PEs, or even programmable general-purpose cores).

1.2.2 SLOT execution model

To address the challenge that prior speculation mechanisms are expensive for accelerators, this thesis also proposes a new execution model, *Spatially Located Ordered Tasks* (SLOT), that does not need cache coherence or explicit tracking of read/write sets. In SLOT, all work happens through tasks that are ordered using timestamps. A task may create children tasks ordered after them, and parent tasks communicate input values to children directly. Each SLOT task *must operate on a single read-write object*, which must be declared when the task is created (besides this restriction, tasks may access an arbitrary amount of read-only data).

SLOT semantics guarantee that all conflicts (tasks that access same read-write data) are localized. Hence, SLOT implementations do not need a coherence protocol. Instead, SLOT allows a *data-centric approach*, where shared data is mapped across the system; work is divided into small tasks that access at most one shared object each; and tasks are always sent to run at the place where their data is mapped. Since each task's read/write set is limited to a single object, SLOT implementations do not need expensive tracking structures such as Bloom filters.

If this execution model was unordered, it would be very limited since it would preclude atomic accesses to multiple objects. However, order can be leveraged to implement atomicity as follows. First, a task that needs to access multiple objects is divided into multiple fine-grained tasks that access a single object each. Then, each group of fine-grained tasks is assigned a non-overlapping set of timestamps to maintain atomicity requirements.

1.2.3 Chronos implementation on FPGA

We first implement a Chronos variant that supports the SLOT execution model. It maps read-write objects across tiles, then sends each created task to the tile where its object id is mapped. This allows low-overhead speculative execution by not requiring a coherence protocol.

We evaluate Chronos by implementing it on an FPGA and use it to implement accelerators for several graph analytics and simulation applications. We use four hard-to-parallelize applications with speculative parallelism. We deploy these accelerators on commodity cloud FPGA instances. We compare these accelerators with state-of-the-art software implementations of these applications running on a higher-priced 40-thread multicore instance. Chronos achieves speedups of up to 15.3× and gmean 5.4× over the software versions. Chronos outperforms the multicore baseline *despite running at a 19× slower frequency*, because it exploits orders of magnitude more parallelism. These results show that FPGAs are a practical and cost-effective way to accelerate applications with speculative parallelism.

1.2.4 Extension of Chronos to support the Swarm execution model

While SLOT simplifies hardware, it suffers from several limitations. First, porting a new application to SLOT requires extra developer effort. The developer needs to break tasks down into small fine-grained tasks that access a single read-write object each, and develop PEs to execute each of these task types. Generally, a small fraction of code dominates execution time, and it is reasonable to expect programmers to carefully map this to SLOT. However, it is hard to map the vast majority of code, and because this code is infrequently executed, it will also yield little benefit.

Second, compared to an equivalent-work coarse-grained task, multiple fine-grained tasks require more on-chip resources to keep track of them. This can result in on-chip structures becoming full frequently, limiting the degree of speculative parallelism.

Third, SLOT programs have performance issues in situations where an object is widely read but is rarely written. For such data, a coherence protocol would enable parallel low-latency reads by distributing the data close to the cores, and invalidate all copies on a write. When writes are infrequent, this is the right approach. But unfortunately, SLOT forces all reads to go through the same place, which causes extra communication, increases serialization, and may also lead to load imbalance.

Therefore, we extend Chronos to support the more general Swarm [54] execution model. In Swarm, similarly to SLOT, all work happens through tasks that are ordered using programmer-defined timestamps. However, unlike SLOT, tasks have no restriction on which data they may access. Swarm allows tasks to access multiple objects.

We call this extension Chronos-Swarm. Chronos-Swarm implements the Swarm execution model through a two-level cache-coherence protocol that is augmented to detect and resolve conflicts. On each memory access, the system aborts all other tasks that accessed the same line with a higher timestamp. However, this comes at the cost of additional latency on memory accesses, and misspeculating tasks running for longer due to delayed conflict checks.

We show that Chronos-Swarm enables efficient acceleration of new applications like transactional processing workloads on tree indices. Compared to SLOT, the Swarm execution model reduces serialization and avoids load imbalance, improving performance by 3.5×.

However, compared to SLOT, Swarm suffers from two drawbacks. First, a similar-sized Chronos-Swarm system takes about 10% more area, limiting the number of tiles that can fitted in a single FPGA. Second, Chronos-Swarm suffers from significant runtime overheads, in extra main memory bandwidth required for conflict detection metadata, and extra conflict checks required per task. We show that these runtime overheads impact performance by up to 2× for applications that naturally map to SLOT.

1.2.5 A hybrid SLOT/Swarm execution model

Given the different tradeoffs between SLOT and Swarm, we propose a hybrid execution model that has the benefits of both. In this hybrid model, a task may follow either SLOT or Swarm semantics. Swarm tasks may access arbitrary objects, but with expensive conflict checks, SLOT tasks limit accesses to a single object, but with object-level low-overhead conflict checks.

This hybrid execution model facilitates incremental application development. The developer may start out with only Swarm tasks. As profiling reveals hotspots, some tasks are refactored to obey the SLOT restrictions and increasing performance.

We extend Chronos to support this hybrid execution model. This implementation, Chronos-Hybrid, can reclaim nearly all performance loss on applications that maps naturally to SLOT. In addition, we show that using the hybrid execution model can achieve performance better than either one alone. In `yscb`, Chronos-Hybrid outperforms the best of the other Chronos variants by 29% by using the Swarm execution model for task that access rarely written data, and the SLOT execution model for the others.

1.3 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 provides relevant background and summarizes related work, including a description of the Swarm execution model. Chapter 3 introduces the SLOT execution model, Chronos framework and the evaluation of Chronos using commodity FPGAs in the cloud. Chapter 4 extends Chronos to support the Swarm execution model. Chapter 5 introduces the hybrid SLOT/Swarm execution model. Chapter 6 concludes the thesis and discusses future work.

CHAPTER 2

Background

Over the past several decades, computer systems have relied on Moore's Law, which predicted that the number of transistors on a chip will double every two years [72], to deliver exponentially increasing performance. However, this trend is expected to slow down as transistors are nearing their physical limits [60]. In addition, since the mid-2000s, power density has become the key limitation of processor performance [32]. Therefore, to keep the performance improvements going, computer architects have now shifted their efforts to making better use of a limited number of transistors.

One promising solution is accelerators, hardware computing engines that are specialized for a particular application or domain. Some accelerators are specialized to a single application, such as a video decoder for a single protocol [1, 3], while others allow for some programmability that allows targeting multiple applications on a particular domain, such as deep learning [23, 57] or graphics [15, 96]. These accelerators often improve performance and energy efficiency by multiple orders of magnitude over general-purpose systems.

In this chapter, we first look at the sources of these efficiencies. We identify four key reasons, and single out one of them, parallelism, for further exploration in this thesis (Section 2.1). We survey the types of parallelism available in applications and how prior accelerators have exploited them (Section 2.2). We observe that one category, speculative parallelism, has largely been unsupported in accelerators. We then describe the utility of speculative parallelism through a representative example (Section 2.3.1), and discuss how prior general-purpose multicores have proposed to exploit it (Section 2.3.2). We identify the limitations in applying similar techniques to accelerators, motivating the need for the SLOT execution model that we present in Chapter 3.

2.1 Sources of efficiency in accelerators

Accelerators achieve significant speedups over general-purpose multicores due to several reasons. We will illustrate these reasons via two accelerators from the deep-learning domain, Eyeriss [23] and Brainwave [25].

Reducing instruction overheads: General-purpose cores spend a significant amount of energy on overheads in fetching and decoding instructions, accessing the register file and performing control actions. In the case of out-of-order cores, these overheads are further exacerbated as they spend energy on finding independent instructions to run, potentially out of program order. For example, consider a simple 32-bit integer add operation. This operation takes only 63 fJ in a 28nm CMOS silicon fabrication technology [48]. But performing an integer add instruction in a 28nm ARM A15 takes 250 pJ [95], a 4000× overhead.

An accelerator can reduce this overhead using several techniques. First, by using coarser instructions, each of which performs many operations, accelerators can amortize instruction fetch and decode overheads. For example, vector instructions, which perform the same operation over multiple data elements, are a common primitive in conventional programmable accelerators, like GPUs [96]. Brainwave, a programmable deep-learning accelerator using a vector ISA, uses this heavily. Brainwave has a single instruction that performs a 128×128 matrix-vector multiplication, amortizing the cost of fetching and decoding a single instruction across 16,384 multiply-and-add operations. Using coarser instructions also removes the need to execute many of them in parallel, as general-purpose processors do to achieve high performance, because more parallelism is extracted from a single instruction.

Second, some accelerators, instead of an instruction-based approach, achieve programmability by relying on rarely changed configurations on a spatial grid of processing elements (PEs). They apply a configuration to these functional units, and computation is performed by streaming data through them. The deep-learning accelerator Eyeriss takes such an approach. In Eyeriss, input data is streamed in through the sides of the grid, and each PE performs a single multiply-accumulate operation and passes the result to the next PE in line. Eyeriss can be configured to support multiple streaming patterns, but once configured, it stays the same for a long period of time, reducing instruction overheads. Coarse-Grained Reconfigurable Arrays (CGRA) [64, 80] also follow the same spatial architecture, but with more programmability within the PE as well as in the connection between PEs.

Finally, fixed function accelerators, like baseband processors on mobile devices [24], completely remove instruction overheads by not allowing any configurability.

Tailoring the accelerator's operation set to the application: Accelerators can create customized datapaths for application-specific operations that are expensive to implement with general-purpose cores. For example, Brainwave has deeply pipelined datapaths for commonly used deep-learning operations like Rectified Linear Unit (ReLU), sig-

moid, and hyperbolic tangent. Implementing the same operations on general-purpose cores is often a multi-instruction sequence.

In addition, accelerators can support reduced-precision arithmetic, leading to lower area and energy consumption over standard-precision (32- or 64-bit) equivalents. For example, Brainwave can be configured to support precisions as low as 5-bits. The resulting area savings can be used support more such functional units.

Tailoring the accelerator to the parallelism inherent in an application: The main strategy that multicores use to exploit parallelism is to run multiple threads in different cores. This is a fairly general technique, but applications often admit more efficient parallelization techniques. In addition, a single core can use more efficient parallelization strategies, like SIMD-style execution using short vector instructions [33]. However, these instructions are limited in how they can be exploited since not all applications can benefit from them.

In contrast, accelerators can customize the type of parallelism exploited in the accelerator to the parallelism inherent in the application. For example, deep learning is highly data parallel, so Eyeriss has 168 PEs all applying the same multiply-accumulate operation on different data elements. Providing this much parallelism on a multicore leads to diminishing returns for applications that do not use it.

Alternatively, accelerators can extract coarse-grained pipeline parallelism when multiple steps depend on each other, but the execution can overlap and the output of one step is streamed as input to the next step. Brainwave uses pipelining to overlap matrix-vector multiplies with data format conversions.

In this thesis, we target applications where speculative execution is needed to to extract sufficient parallelism to fully utilize available execution units. Chronos provides a reusable framework to extract task-level parallelism. In addition, it allows application developer to exploit intra-task parallelism, using either data- or pipeline- parallel techniques, by by supporting different styles of PEs.

Tailoring the accelerator's memory hierarchy to the application: The gains from specialization and parallelism are dependent on keeping the computation supplied from small, local memories, such as scratchpads or queues. Compared to general-purpose systems, which feed data from a rigid cache hierarchy, these small local memories can sustain lower latency, higher bandwidth, and can be specialized to application requirements.

For example, each PE in Eyeriss has a small local memory to store locally used data, and uses on-chip FIFO queues to communicate across PEs.

2.2 Types of parallelism exploited in accelerators

In this section, we survey the types of parallelism available in applications and how prior accelerators have exploited them. We use a task-based model to illustrate the different types of parallelism.

In a task-based model, the programmer expresses the computation as a set of tasks. Tasks allow decoupling logical parallelism from physical parallelism. Each task is a unit of computation work, and the number of tasks may be far larger than the number of processing elements.

We classify prior accelerators by the type of parallelism they target. We establish two broad categories based on whether the tasks and their dependences are known in advance. If so, then we consider the application to have static parallelism. Otherwise, we consider the application to have dynamic parallelism.

2.2.1 Static parallelism

Applications have static parallelism when their tasks and the data dependences among them are known ahead of time. Static parallelism commonly arises in applications that manipulate dense matrices and arrays, and have predictable control flow (e.g., matrix-matrix multiplication). These applications typically comprise loops where each iteration performs the same operation on a different data element. If the iterations are independent of one another, then they may execute in parallel. Static parallelism also manifests in applications with pipeline parallelism. Here, computation is partitioned into a sequence of pipeline stages, with each stage producing data consumed by the following stage. Stages may execute in parallel, communicating shared data in deterministic producer-consumer style. This style is common in video compression pipelines [1].

Applications with static parallelism are analyzable in advance. Therefore, they are amenable to a large number of compile-time techniques, such as points-to analysis [83] and shape analysis [66]. In software, there have been auto-parallelizing compilers that take a sequential program, analyze it using these techniques, and generate a parallel implementation [7, 14]. In hardware, High-Level Synthesis (HLS) tools [16, 58] use similar techniques to transform a description of an application written in a high-level language down to a high-performance circuit description.

Frameworks: A common organization for accelerators extracting static parallelism is to use a systolic array style structure [59]. A systolic array consists of a grid of processing elements (PEs). A PE may consist of an ALU and a small amount of memory. Each PE computes a function of the inputs it receives from some neighboring PEs, and stores the result within itself or passes it to other neighboring PEs.

This framework of a grid of PEs is an ideal match for mapping applications with dense matrix operations, and has been used in deep neural network accelerators such as Google's TPU [57] and Eyeriss [23]. Systolic arrays have also been used in genomics applications [94] to accelerate the Smith-Waterman algorithm, and in a variety of signal processing algorithms [30, 49, 97].

2.2.2 Dynamic parallelism with independent tasks

Applications have dynamic parallelism when their tasks or their dependences are unknown in advance. Because this information is not known until runtime, the dependences in these application cannot be handled at compile time. Therefore, we need runtime support to dynamically handle tasks as they are created and executed.

We distinguish three different kinds of dynamic parallelism based on how much synchronization they need.

No synchronization with independent tasks

First, we consider the simplest case when the tasks operate on disjoint read-write data and need no synchronization for shared data accesses. This kind of structure is common in applications that traverse sparse data structures, such as trees and graphs.

For example, consider an iterative, pull-style graph application, like PageRank [9]. In such an application, each task operates on a single node. The task reads neighbors' property values from the previous iteration, performs some computation on them, and updates the node's own property value for the current iteration. Within an iteration, no two tasks operate on the same node, so no two tasks access shared read-write data, and therefore need no synchronization. In addition to graph accelerators that use this model [74, 103], sparse-matrix accelerators like EIE [43], Gamma [100], and SCNN [77] also fall under this category.

These accelerators follow an architecture consisting of multiple independent PEs, with some local storage. The allocation of tasks to PEs could be either static or dynamic. In a static mapping, tasks are allocated to PEs based on some parameter known at task creation time. For instance, in EIE, each task operates on a row of a sparse matrix, and each task operating on the same row is mapped to same PE. However, different rows may have different numbers of non-zero elements, so the amount of work per task can vary significantly. Under static mapping, this might lead to an unequal distribution of work, with some PEs being stalled while the others are busy. Handling this situation, known as *load balancing*, is the primary concern among most accelerators that follow this model.

Dynamic task mapping schemes can mitigate this load imbalance issue. Some accelerators, such as EIE, maintains a task queue for each PE. A central scheduler looks at the occupancies of these queues and dispatches tasks to the one with the lowest occupancy. Alternative load balancing techniques like work-stealing [34] could also be used. In work-stealing, tasks are assigned statically to PEs, but if a PE runs out of tasks, it steals some from another PE's task queue.

Frameworks: Frameworks that support dynamic parallelism, like GraphGen [76] and GraphPulse [81], provide a hardware template that provide task queues and a scheduler to perform load balancing. New applications can customize this template by creating PEs to express application-specific logic.

Non-speculative synchronization of dependent tasks

When tasks can operate on shared data, we need synchronization techniques to maintain correctness and atomicity. When tasks access arbitrary data, this is often done through shared synchronization variables, such as locks.

Graphicionado [40] and Li et al. [62] are accelerators for graph algorithms that implement push-style graph processing. Here, a task performs atomic read-modify-write operations on a node's neighbors. Ensuring atomicity when multiple tasks access the same data is achieved through locking. A lock is a shared synchronization variable that allows a task to retain exclusive access to an object. These accelerators implement locks through small on-chip associative memories, which store the set of locks all tasks have currently acquired. When a task wishes to acquire a lock, it queries this memory to see if the lock is already acquired by another task. If not, the task acquires the lock by writing into the memory, performs the necessary updates to the object, and releases the lock by removing the memory entry. If the lock cannot be acquired, the task is stalled until the lock becomes available.

Non-speculative synchronization is also used when tasks create other tasks. Sometimes, the parent may need to communicate with the child task. If the parent is the producer and the child is the consumer; handling this is trivial; the parent simply passes the value to the child as a task argument.

In the opposite case, where the child returns a value to the parent, the parent needs to wait until the child completes. This is a special case of the more-general scenario where the consumer task may be ready before the producer has produced it. In this case, the system needs to ensure that the consumer does not proceed until all of its dependent tasks have completed. This is an example of *dataflow* execution [11,29,34].

Anton [86] implements dataflow execution by using a shared synchronization variable to determine when a task can start. Each task has an associated counter variable, which is initialized to the number of tasks that must be completed before this task can start. When a predecessor task finishes, it atomically decrements this counter. When this counter reaches zero, the task has met all dependences and can now be executed.
Frameworks: TAPAS [69], and ParallelXL [22] are frameworks that allow accelerator development that support dataflow execution. TAPAS is an HLS toolchain that uses a compiler pass to generate tasks complying to the fork-join model [11,34], where tasks can create new parallel tasks and wait for them to finish before continuing execution. TAPAS also generates the PEs for these tasks and provide hardware mechanisms for task scheduling and synchronization.

TAPAS limits synchronization to parent-child tasks. However, ParallelXL supports more general Anton-like synchronization of any two tasks. Each ParallelXL task is provided with a *continuation*, an index to another task whose synchronization variable is decremented when the first task finishes. ParallelXL also provides a reusable hardware template that implements this form of synchronization and allows for customization of the PEs that implement task logic.

Speculative synchronization of dependent tasks

In speculative or optimistic parallelization, the system executes tasks in parallel assuming that their dependences are not violated. The system software or hardware automatically detects dependence violations and recovers from them. If there is no violation, the tasks are committed. In case of a violation, the offending tasks are aborted and re-executed.

Since speculative execution is the core technique of this thesis, we take a detailed look at it in the next section.

2.3 Speculative parallelism

In this section we illustrate how speculation can reveal otherwise hard-to-extract parallelism through a simple application, discrete event simulation (`des`). We then a discuss the Swarm execution model, which extracts this parallelism in a general-purpose multicore setting. We describe how implementing Swarm adds overheads that are expensive on accelerators, motivating the similar, but simpler SLOT execution model (Chapter 3).

2.3.1 A case for speculative parallelism

We illustrate the utility of speculative parallelism through `des`, a discrete event simulator for digital circuits [46]. Listing 2.2 shows code for a sequential implementation of `des`. Each `des` task processes a gate input toggling at a particular time. If this input toggle causes the gate's output to toggle, the task enqueues events for all inputs connected to that output at the appropriate times. The sequential implementation processes one task at a time in simulated time order, and maintains the set of tasks to process in a priority queue.

Figure 2-1a shows a circuit with input waveforms and propagation delays, and Figure 2-1b shows the task diagram of an execution of `des` on this circuit. Arrows between tasks show parent-child dependences (e.g., task O1 creates tasks A4 and X6). The x-axis shows task order, and the task's location in the y-axis represents the gate it operates on.

Parallelism exists despite order constraints because independent tasks may run out of order. In our `des` example, only tasks that operate on the same gate have a data dependence; others (e.g., O1 and N2) may run out of order without violating correctness. But tasks and dependences are not known, so running tasks out of order is not straightforward.

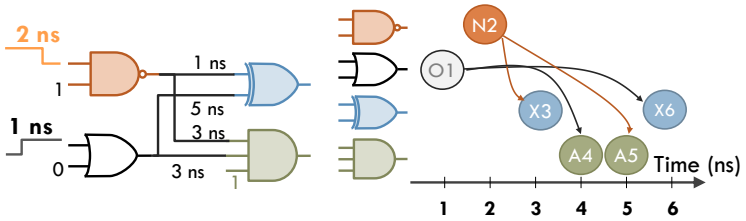
In `des`, each task operates on a single object (a gate), and this object is known in advance. But this is not sufficient to find which tasks are safe to run, because *a task may have a dependence with another task that comes earlier in program order but does not yet exist*. Suppose that task O1 is executed first, producing task X6. At this point, X6 is the earliest task in the system that operates on the XOR gate. But executing X6 would

```
PrioQueue<Time, GateInput> eventQueue;

void simToggle(Time time, GateInput input) {
 Gate gate = input.gate;
 bool outToggled = gate.simulateToggle(input);
 if (outToggled) {
  // Toggle all inputs connected to this gate
  for (GateInput i : gate.connectedInputs()) {
   Time nextTime = time + gate.delay(input, i);
   eventQueue.enqueue(nextTime, i);
}}}

... // Enqueue initial events (input waveforms)
// Main loop
while (!eventQueue.empty()) {
 (time, input) = eventQueue.dequeue();
 simToggle(time, input);
}
```

Listing 2.1: Sequential implementation of `des`.



(a) Example circuit (b) Ordered tasks produced by *des*

Figure 2-1: Example execution of `des`: (a) Input circuit and (b) Ordered tasks produced by the execution of `des` on this circuit. Tasks O1 and N2 correspond to inputs of the NAND and OR gates; both tasks toggle their gates' outputs, producing tasks X3, X6 and A4, A5.

produce incorrect results, because X6 must follow the earlier data-dependent task X3, which does not yet exist (as N2 has not been run).

Some prior work in parallelizing `des` have relied on non-speculative approaches like the Chandy-Misra-Bryant (CMB) algorithm [20]. The CMB algorithm maintains a task queue (FIFO) for each input of each gate. The algorithm reasons that if each input of a gate has at least one pending task, the lowest time tasks among all inputs is safe to be executed.

Sometimes, processing a task does not toggle the output of a gate, e.g., an OR gate with a 1 and 0 as current inputs does not toggle when the 0 input is changed to a 1. Ideally, tasks that do not toggle the output should not create new tasks for the fanout gate. But [20] shows that doing so can cause deadlocks, where no gate in the circuit has pending tasks on all its inputs. For example, assuming the output of OR gate is connected to one of the inputs of an XOR gate (as in Figure 2-1a), not creating a task for the XOR gate would prevent any tasks on its other input from being executed.

The CMB algorithm argues that the only way around this restriction is to create

a new tasks for every fanout gate, even if the gate output did not toggle. This would ensure each input in the downstream gate has at least one task, preventing execution of tasks on its other inputs being stalled. Unfortunately, this scheme leads to a significant increase in the number of tasks and substantial communication. If the activity factors are low, i.e., most task do not toggle the output, then most of this communication is ineffectual. Most tasks simply indicate the lack of change on the inputs.

A more efficient approach is to run tasks in parallel, speculating that, for each task, no earlier data-dependent tasks will exist. If speculation is correct, the task can commit and the system has successfully elided order constraints; but on an order violation, the misspeculating task and its descendants need to be aborted and re-executed in the right order. Under this scheme, a task would only creates new tasks if the gate's output toggles as a result of the new input.

This execution strategy, known as Time Warp in the case of `des` [51], shows that speculation arises from the need to preserve order constraints, *even though the data that each task accesses is known in advance*. In Chapter 3, we will see that advance knowledge of task data accesses enables a simpler implementation of speculative execution, using the SLOT execution model.

When to speculate: In general, speculation is desirable when undoing work is unlikely and finding independent work is too expensive relative to the cost of keeping track of that work.

In the case of `des`, prioritizing earlier tasks when there are multiple pending tasks makes out-of-order dependent tasks sufficiently rare, and thus ensures that undoing work is fairly unlikely. In addition, providing hardware support for keeping track of memory accesses would ensure that the overheads of speculation remain tractable. Our framework, Chronos, ensures both by ensuring tasks are dispatched in priority order and offering hardware support for conflict detection and resolution.

The usefulness of speculation depends on the characteristics of the input. Speculation cannot extract parallelism if the circuit only has a single path (e.g., a chain of inverters). Speculation is also unnecessary when simulating large parallel circuits with many independent paths (e.g., 64-bit bitwise AND operator) where finding independent work is cheap. Speculation is only helpful when there are many dependent paths (e.g., adder circuits).

2.3.2 Execution models for speculative parallelism

Speculative parallelism can be extracted using different execution models.

Thread-Level Speculation (TLS) [37,41,84,89,91,101], allows multiple threads to execute tasks of a single sequential program, such as loop iterations, in parallel while preserving the appearance of sequential execution. TLS executes the first iteration of a loop non-speculatively and each iteration spawns a speculative task to run the next iteration. The system tracks the memory addresses accessed by each task, detecting

conflicts when two accesses to the same location happen in wrong order and at least one them is a write.

Hardware Transactional Memory (HTM) [4, 13, 19, 36, 44, 47, 50, 73]. allows programmers to declare certain code blocks as transactions, while guaranteeing that the transactions execute in an atomic and isolated manner with respect to other code blocks. Similar to TLS, TM implementations track memory addresses and perform conflict detection. However, unlike in TLS, there are no ordering requirements; any ordering of transactions is considered to be valid.

Recently, we have proposed a new execution model based on speculative ordered tasks [2, 53, 54, 55, 93, 99]. This model, which we named Swarm, expresses a program as a set of ordered tasks. Each task is assigned a programmer-defined integer, called a timestamp, to specify its order with respect to the other tasks. This model subsumes both TLS and TM; TLS can be emulated by tasks with contiguous timestamps, while TM can be emulated with tasks with equal timestamps. In addition, Swarm can express a much more broader set of algorithms, including `des`, by supporting explicit ordering requirements.

Due to this generality, this thesis targets the speculative ordered tasks model. Therefore we explain the Swarm execution model in detail next. In Chapter 3, we propose a new execution model, SLOT that is derived from Swarm but adds restrictions to facilitate a simpler coherence-free implementation.

2.3.3 Swarm execution model and multicore implementation

Swarm programs consist of timestamped tasks. Each task may access arbitrary data, and can create child tasks with any timestamp greater than or equal to its own. Swarm guarantees that tasks appear to run in timestamp order. If multiple tasks have equal timestamp, Swarm chooses an order among them.

Swarm exposes its execution model through a simple API. Listing 2.2 illustrates this API by showing the Swarm implementation of `des`,

Each task runs a function that takes a timestamp and an arbitrary number of additional arguments. Listing 2.2 defines one task function, `desTask`, which simulates a signal toggling at a gate input. Tasks can create child tasks by calling `swarm::enqueue` with the appropriate task function, timestamp, and arguments. In our example, if an input toggle causes the gate output to toggle, `desTask` enqueues child tasks for all the gates connected to this output. Finally, a program invokes Swarm by enqueuing some initial tasks with `swarm::enqueue` and calling `swarm::run`, which returns control when all tasks finish. For example, Listing 2.2 enqueues a task for each input waveform, then starts the simulation.

Swarm's execution model generalizes TLS by *decoupling task creation and execution orders*: whereas in TLS schemes a task can only spawn speculative tasks that are immediate successors [41, 42, 85, 89, 91], a Swarm task can create child tasks with any timestamp equal or higher than its own. This allows programs to convey new work to

```
void desTask(Timestamp ts, GateInput* input) {
  Gate* g = input->gate();
  bool toggledOutput = g.simulateToggle(input);
  if (toggledOutput)
    // Toggle all inputs connected to this gate
    for (GateInput* i : g->connectedInputs())
      swarm::enqueue(desTask, ts+gate.delay(input, i), i);
}

void main() {
  [...]  // Set up gates and initial values
  // Enqueue events for input waveforms
  for (GateInput* i : externalInputs)
    swarm::enqueue(desTask, 0, i);
  swarm::run();  // Start simulation
}
```

Listing 2.2: Swarm implementation of discrete event simulation for digital circuits.

hardware as soon as it is discovered instead of in the order it needs to run, exposing a large amount of parallelism for ordered irregular applications [54].

Swarm implementation

Swarm uncovers parallelism by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest active (unfinished) task. Swarm introduces modest changes to a tiled, cache-coherent multicore (Figure 2-2). Each tile has a group of cores, each with its own private L1 cache. All cores in a tile share an L2 cache, and each tile has a slice of a fully-shared L3 cache. Every tile is augmented with a *task unit* that queues, dispatches, and commits tasks. Swarm hardware efficiently supports fine-grain tasks and a large spec-

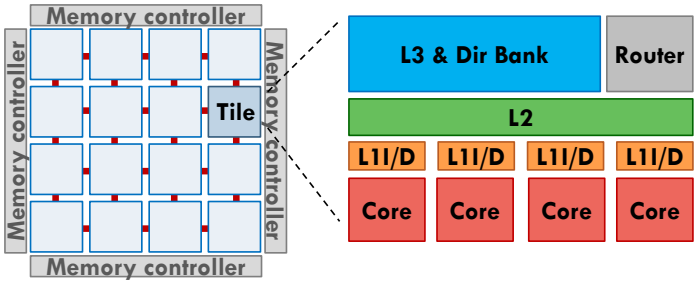


Figure 2-2: Structure of Swarm multicore implementation

ulation window through four main mechanisms: low-overhead hardware task management, large task queues, scalable data-dependence speculation mechanisms, and high-throughput ordered commits [54].

Chronos adapts these techniques to accelerators. Compared to general-purpose systems, accelerators have higher throughput. Next, we discuss how cache coherence affects this throughput.

2.3.4 Prior speculative architectures rely on cache coherence

Swarm, and other prior architectures for speculative parallelization, extend a cache-coherent multicore. These systems *reuse existing mechanisms* to implement speculative execution. Specifically, they adapt the cache coherence protocol for conflict detection.

Coherence-based conflict detection works by leveraging invalidation and downgrade messages to detect conflicts. Each task runs in a single core. The core acquires coherence permissions for each read and write as usual, but keeps permissions for these lines throughout the execution of the task (either by keeping the task's data in the private cache [4, 19, 89], or by tracking these permissions in the shared directory [54, 73, 98]). Thus, the core will receive an invalidation or downgrade request on every possible conflict (i.e., if another task issues a read to a line in the task's write-set, or any access to a line in the task's read- or write- set).

While relying on coherence is reasonable for multicores, it is expensive for accelerators. Accelerators in general and reconfigurable hardware in particular do not have an coherent cache hierarchy that supports invalidation-based conflict detection [87]. Implementing coherence would add complexity, latency, and significant on-chip SRAM to implement a directory that tracks sharers. To understand the effects of these overheads, we first develop SLOT (Chapter 3), an execution model that does not rely on cache coherence, and implement it on the Chronos framework. In Chapter 4, we extend our implementation to Swarm which facilitates a quantification of these overheads.

2.3.5 Accelerators targeting speculative parallelism

There have been limited prior work in applying speculation to accelerators. We review some of these accelerators now.

Transactional Memory: Ma et al. [67] proposes a framework that allows acceleration of applications using Transactional Memory. Similar to Chronos, it consists of a reusable framework for supporting application independent operations, and allows customization of PEs for different applications. However, this framework does not support caches. It serves all memory requests from main memory, incurring large latency on memory accesses and increasing the probability of conflicts.

Graph accelerators with priority scheduling: Some frameworks such as PolyGraph [27] and Ma et al. [67] support priority scheduling when generating accelerators for graph algorithms. These frameworks associate each task with a priority and dispatches tasks to PEs in this priority order. However, unlike Chronos, these frameworks cannot always enforce this priority, and sometimes allows tasks to run out of order. Therefore, they can only support applications whose tasks are naturally resilient to out-of-order execution.

Simulation accelerators: Prior work in parallel discrete event simulation has proposed accelerators for different aspects of task-level ordered speculation. The Rollback chip [35] accelerates the speculative versioning and rollback process, but leaves other

aspects such as conflict detection to software. Rahman et al. [82] implement a discrete event simulation accelerator on an FPGA. This uses a centralized design that uses only a single event queue. This single event queue saturates at around 0.15 events per cycle, a 50× lower task throughput than a 16-tile Chronos system which can support 2 events per cycle per tile. Moreover, Rahman et al. evaluated their design using a microbenchmark with long tasks and do not explore how to accelerate actual applications. Hence, they do not consider subtle issues that arise when doing so, such as dealing with limited on-chip queue capacity.

CHAPTER 3

Chronos: Efficient Speculative Parallelism for Accelerators

The goal of this thesis is to provide a framework that makes it easy to build accelerators that leverage speculative parallelism. But as we described in Chapter 1, prior exection models that extracted speculative parallelism can be expensive on accelerators. Therefore, we need a new execution model that is tailored for use in accelerators.

In this chapter, we first propose a new execution model, Spatially Located Ordered Tasks (SLOT), that does not need cache coherence to support task-level speculation. We then describe the Chronos framework, which implements this SLOT execution model. Chronos provides a reusable accelerator template that provides support for task management and speculative execution. Accelerator designers can then specialize this template by defining their own PEs.

We evaluate Chronos by implementing it on FPGAs and using it to generate accelerators for four challenging applications. We compare the performance of these accelerators in comparison to highly-optimized software parallel baselines for the same applications. Even with the FPGA's 19× slower frequency, these accelarators are 3.5×-15.3× faster than the baselines.

3.1 The SLOT Execution Model

We now present the *Spatially Located Ordered Tasks* (SLOT) execution model. SLOT restricts each task to access a single read-write *object*, which must be known when

the task is created (Section 3.1.2). To motivate this requirement, we first look at why speculation is needed in the first place.

3.1.1 Reasons for speculative execution

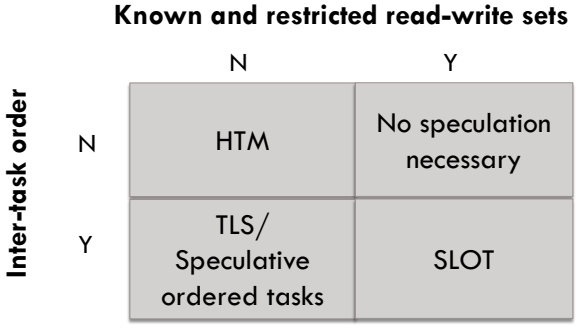


Figure 3-1: A categorization of the reasons for speculative execution. Cache coherence is only required if read-write sets are unknown or unrestricted. We will show that inter-task order is sufficient for all aplications.

In general, speculation is needed when tasks have *either* unknown read- and write-sets *or* inter-task order constraints.

By relying on coherence, prior speculative systems support tasks with unknown read- and write-sets. Speculation allows HTM systems to preserve atomicity among unordered tasks (transactions), and TLS systems to enforce both atomicity and order among tasks. But as we saw in Section 2.3.1, `des` needs speculation to elide order constraints among tasks even though each task's read/write-set is known in advance.

Figure 3-1 presents systems according to their reasons for speculative execution. As we can see, prior architectures all support tasks with unknown read- and write-sets, which forces complex conflict detection. In SLOT, we focus on the remaining quadrant: supporting *inter-task order* only, but where tasks have *known and restricted read- and write-sets*. This simplification is sufficient for `des`; in Section 3.1.3) we will see that inter-task order is in fact sufficient to support unknown read- and write-sets, because *inter-task order allows breaking work into tasks with known read- and write-sets*.

3.1.2 Spatially Located Ordered Tasks

SLOT applications consist of ordered, dynamically created tasks. Each task can be implemented in software or hardware. We describe the execution model independently of the implementation, and illustrate it using the software API.

Each task is given two attributes when it is created: a *timestamp* and an *object id*. Timestamps specify order constraints: the system guarantees that tasks appear to execute in timestamp order. Tasks with equal timestamps may run in any order, but are atomic (i.e., they do not interleave).

```
void simToggle(Time time, GateInput input) {
 Gate gate = input.gate;
 bool outToggled = gate.simulateToggle(input);
 if (outToggled) {
  // Toggle all inputs connected to this gate
  for (GateInput i : gate.connectedInputs()) {
   Time nextTime = time + gate.delay(input, i);
   slot::enqueue( simToggle,   // task type
                  nextTime,    // timestamp
                  i.gate.ID,   // object id
                  i  );        // args
}}}
```

Listing 3.1: SLOT implementation of `des` task.

Object ids are integers that specify the data dependences among tasks: two tasks are treated as data-dependent *if and only if they have the same object id*. Object ids restrict each task to accessing at most one read-write object in shared memory. Note that this restriction only applies to read-write data. A task may access any amount of read-only data.

A SLOT task can create children tasks as it finds more work to do, by specifying the type of the child task, as well as its timestamp, object id, and any input data values it may need. Each child task may have any timestamp that is greater than or equal to its parent's.

In SLOT, parent-child relations are *unidirectional*: a parent task can create and pass values to its children, but parents are ordered before their children and thus appear to complete before children execute. Child tasks cannot return values or communicate with their parents. This is different from fork-join execution models like Cilk [34], where parents wait for their children to complete.

API: Listing 3.1 illustrates the SLOT software API by showing the implementation of a `des` task. In software, each task is implemented by a function. The implementation is almost the same as the sequential one in Listing 2.2: each task simulates an input toggle at a particular gate. Instead of enqueuing tasks to a priority queue, this code creates new tasks by calling `slot::enqueue`, which specifies the child task's type (its function pointer since it's a software task), timestamp, object id, and any additional arguments (the gate input in this case). We will later describe the hardware interface for specialized SLOT PEs.

SLOT enables coherence-free conflict detection: By restricting each task to access at most one read-write object, implementations of SLOT can perform distributed conflict detection without complex tracking structures. If the implementation maps object ids across cores or tiles, and sends each task to where its object id is mapped, then finding conflicts becomes a local operation.

For example, if Figure 2-1 was run on a four-core system, the NAND, OR, XOR, and AND gates could be mapped to cores 1–4. Then, if task X3 arrives in core 3 after

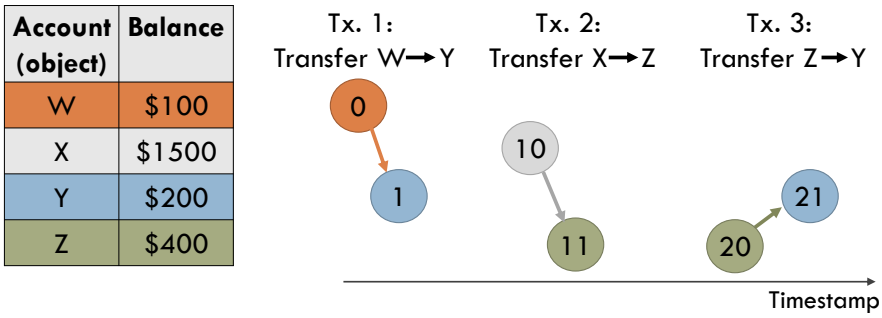


Figure 3-2: Example showing how to leverage order to implement atomic accesses to multiple read-write objects. Each transaction is broken down to multiple tasks that access one object each. Atomicity is maintained by assigning a disjoint timestamp range to each transaction.

X6 has already run, core 3 can determine X3's conflicts (tasks for the same gate and a higher timestamp, {X6} in this case) locally, by comparing X3's object id with those of still-speculative tasks.

3.1.3 Mapping multi-object computations to SLOT

While SLOT's single-object restriction is natural for applications like `des`, many applications must perform atomic accesses to multiple read-write objects, and may not know all these objects in advance.

Nonetheless, SLOT's *support for order* enables a trivial, systematic mapping of these computations to SLOT. Specifically, multi-object transactions can be expressed as SLOT tasks by breaking each transaction into multiple single-object tasks, each accessing a single object, and *giving each transaction a disjoint range of timestamps*. This way, tasks within a transaction do not overlap with those in other transactions.

For example, consider a banking application where transactions transfer money between accounts. Each transaction must atomically decrement the source account's balance and increment the destination account's balance. To scale, each account should be a different object; but since account balances are read-write data, a single task cannot access two accounts.

Figure 3-2 shows how task order makes this possible. We implement each transaction using two SLOT tasks, each of which manipulates a single account: the first decrements the source's balance and creates a second task to increase the destination's balance. Each transaction has a disjoint range of timestamps, so tasks from different transactions do not interleave.

This technique generalizes to arbitrary combinations of read-write operations. For example, our implementation of `maxflow` (Section 3.3) uses it to perform complex atomic operations on the neighborhood of a graph vertex.

While breaking each transaction into many small tasks could add significant overheads to a software runtime, small tasks are a natural match for an accelerator, as hard-

ware performs task management and small tasks need simple processing elements.

3.1.4 Discussion

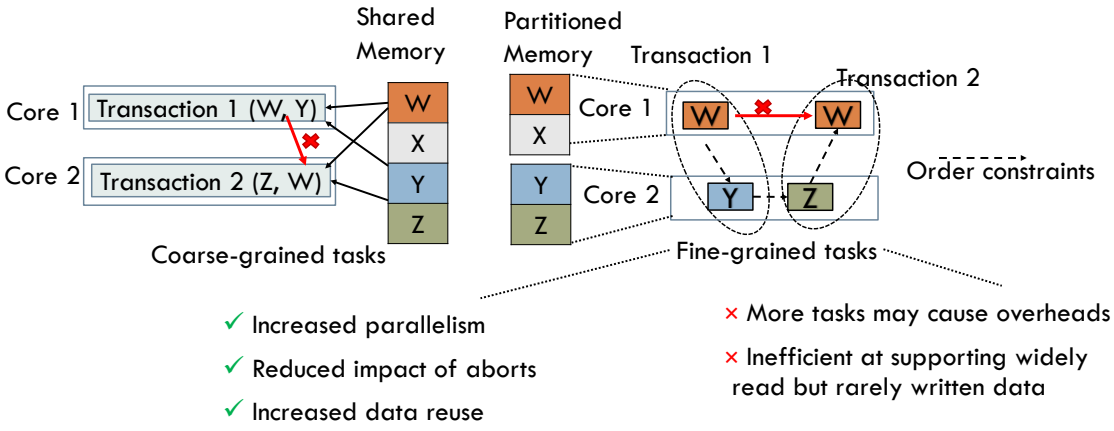


Figure 3-3: Benefits and limitations of fine-grained tasks

Benefits of SLOT's fine-grained tasks: SLOT's key advantage over prior work is to enable coherence-free conflict detection. In addition, prior work [53, 93] has shown that, even in systems that support tasks with arbitrary read/write-sets, this division is often desirable, for three key reasons:

(i) Increased parallelism: Breaking a long serial transaction into short tasks allows these tasks to run in parallel.

(ii) Reduced impact of aborts: On misspeculation, only the tasks that conflict are aborted, rather than the entire transaction. For example, as illustrated in Figure 3-3, if a conflict arose between the two coarse-grained tasks over the object W, then transaction 2 has to be aborted in its entirety. In contrast, with fine-grained tasks, only the small task that accessed W would be aborted.

(iii) Increased data reuse: Rather than bringing shared data across the system where the transaction is running, tasks are sent to run close to their data, avoiding cache line ping-ponging. Since each task message is much smaller than a cache line, this reduces traffic; and tasks are sent and executed asynchronously, so their latency is easier to hide than that of synchronous memory accesses.

SLOT limitations: While breaking programs into short single-object tasks is generally beneficial, there are several situations where this would be unproductive.

(i) Requires more on-chip resources: Fine-grained tasks requires more on-chip task storage resources. Depending on the structure of these tasks, this may end up causing overheads.

To understand why, we refer to *Little's Law* [63], which states that the long-term average number L of customers in a stationary system is equal to the long-term average effective arrival rate λ multiplied by the average time W that a customer spends in the system.

Applying this to tasks, breaking down a coarse-grained task into multiple fine-grained tasks decreases L. If this breakdown does not expose more parallelism than the original task, then W stays constant. This results in a reduction of λ, the throughput of coarse-grained tasks.

(ii) Inefficient at supporting widely read but rarely written data: If the application is dominated by *rarely modified* read-write data that has substantial reuse, coherence-based conflict detection would allow caching this data across the system, making reads between the sporadic writes local, whereas SLOT needs to isolate each access to these data in a separate task and send them to a single place. As we will see in Chapter 4, this causes extra communication, increases serialization, and may also lead to load imbalance.

In this chapter, we target applications where SLOT's benefits outweigh its limitations. In Chapter 4, we consider applications where the limitations outweigh the benefits.

Relationship with prior work: Spatial hints [53] and Espresso [55] also propose to tag tasks with an identifier similar to an object id, but with different goals. Spatial hints are used to distribute speculative tasks so that tasks *likely* to access the same data run at the same place. But spatial hints are optional and advisory, and the system must still use coherence-based speculation. Espresso uses locales to additionally provide mutual exclusion among *non-speculative* tasks. By contrast, SLOT requires all tasks to specify correct object ids, i.e., to identify the read-write object that they will access. This enables using object ids to implement conflict detection among speculative tasks.

3.2 Chronos Framework

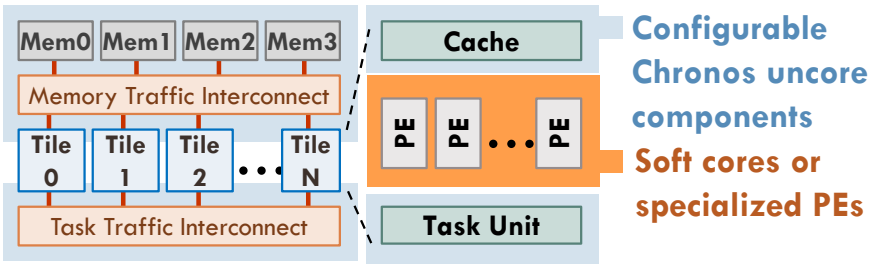


Figure 3-4: Chronos framework overview.

Chronos is an architectural framework that makes it easy to design accelerators for applications with ordered parallelism. Chronos achieves this by providing an *architecture template* that implements the SLOT execution model efficiently. Accelerators can then specialize this template by defining their own task processing engines or configuring Chronos's uncore components. With this division, creating a Chronos accelerator is as simple as specifying the processing engines; the framework takes care of the intricacies of ordered task management and speculative execution.

Figure 3-4 shows Chronos's organization. Chronos is a tiled design with fully distributed task management and speculation mechanisms. Each tile has several Processing Elements (PEs) that execute tasks, a local (non-coherent) cache, and a *task unit* that queues, dispatches, and commits ordered tasks.

3.2.1 Design requirements and techniques

Chronos must run short ordered tasks efficiently. This requires achieving *high-throughput task management* and a *large speculation window*:

1. High-throughput task management: Short tasks place high throughput demands on the system. For example, if each task takes 20 cycles to execute, a Chronos system with 200 PEs must create, dispatch, conflict-check, and commit 10 tasks per cycle to keep the PEs busy. This forces a design *without centralized components*: all task management and speculation mechanisms must be fully distributed. Chronos's tiled design achieves this. Moreover, each tile's task unit needs to maintain a high throughput as well.

2. Large speculation window: To prevent order from limiting parallelism, the system must be able to speculate far ahead of the earliest unfinished task. More specifically, due to order restrictions, tasks may stay speculative for a long time before they can commit—far longer than the time they take to execute. Therefore, the system should be able to track many more speculative tasks than running tasks. For example, as we will see in Section 3.4, some applications require about 10 speculative tasks per running task.

These requirements force *fully distributed, deep out-of-order execution*. To achieve these requirements, several of Chronos's techniques are adapted from Swarm [54]. Specifically, Chronos borrows Swarm's *task management* and *ordered commits* techniques. However, Chronos implements speculative execution differently, by leveraging the SLOT execution model instead of relying on a coherent cache hierarchy. We first describe how Chronos performs speculative execution, then detail its task management structures.

3.2.2 Distributed ordered speculation

Chronos uses speculative execution to elide order constraints. Chronos can run any task as soon as it is created, even if its ancestors are still speculative. Figure 3-5 shows the

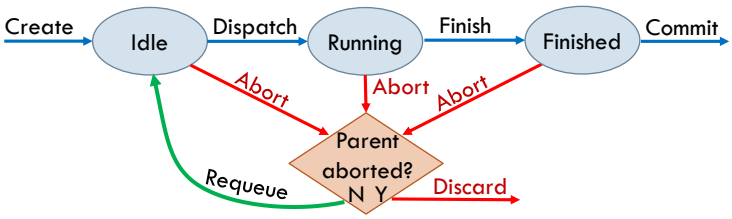
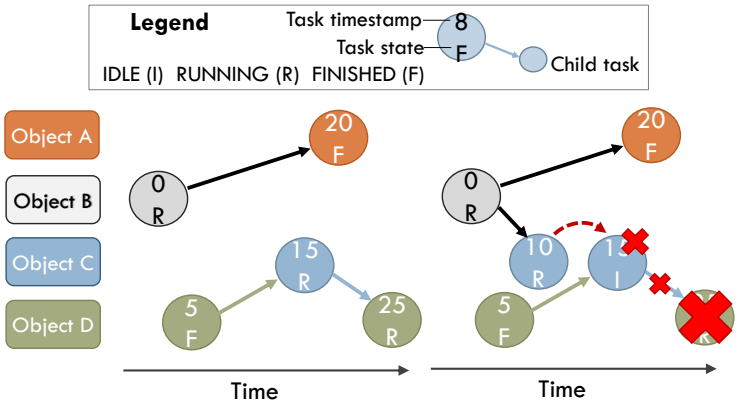


Figure 3-5: Task life cycle.



(a) Tasks can run out of timestamp order

(b) Selective aborts: Task 10 aborts 15 and 25, but not 20

Figure 3-6: Chronos uncovers parallelism by running tasks out-of-order (a). On a conflict, Chronos aborts are selective: they only affect the mispeculating tasks (b).

execution flow of each task. Top horizontal arrows denote correct speculation. When a task is created, it is sent to a tile where it stays *idle*, queued until it is ready to dispatch. The tile dispatches idle tasks to PEs in timestamp order. After a *running* task finishes execution, it stays speculative (in the *finished* state) until the system determines it is safe to commit.

Figure 3-5 shows that tasks may be aborted at any point before commit. Because tasks may run while their ancestors are still speculative, aborting a task requires aborting *and discarding* all its descendants. These cascading aborts are necessary to uncover parallelism, and are *selective*: aborts undo the effects of the aborting task, its descendants, and any data-dependent tasks that come later in program order. As shown in Figure 3-5, if a task is aborted because its parent has aborted, then it is discarded; otherwise, the abort is due to a data dependence, the task is requeued for execution.

Figure 3-6 shows an example of speculative execution in Chronos. Tasks are created and run out of order: in Figure 3-6a, task 20 has run and finished even though earlier tasks are still running; in particular, task 0, 20's parent, is still running. In Figure 3-6b, task 0 creates a child with timestamp 10, which conflicts with task 15. This causes 15 to be aborted, along with its child task 25. Though aborts may affect multiple tasks, they are selective: independent tasks such as 20 are not aborted.

Task mapping and conflict detection: To perform speculative execution cheaply, Chronos

uses the task mapping and conflict detection strategy outlined in Section 3.1: Chronos maps object ids across tiles, then sends each created task to the tile where its object id is mapped. Our current Chronos implementation uses a static object-to-tile mapping: the object id is simply hashed to produce the tile id. We find that this achieves good load balance in our workloads; Chronos could also adopt more sophisticated load balancing based on dynamic remapping of objects among tiles [53].

Task dispatch: The task unit dispatches tasks to PEs in timestamp order to prioritize earlier tasks. To avoid conflicts, the task unit serializes the execution of tasks with the same object id. Therefore, conflicts among running tasks never arise; only a task that arrives to a tile out of order can create a conflict.

Speculative value management: Chronos adopts eager version management: speculative writes update memory in-place, and old values are written to a separate *undo log*. Commits are fast, as the undo log is simply discarded; aborts require restoring the old values from the undo log.

Eager version management facilitates running chains of data-dependent tasks without waiting for them to commit: if task *A* writes a value that is later read by (same-object) task *B*, *B* will naturally use *A*'s value even when *A* has not yet committed. This process, known as *speculative forwarding*, is important for ordered speculation [55], but would be hard to do with lazy version management.

High-throughput commits: To determine when a task can commit, Chronos borrows the Global Virtual Time (GVT) protocol from prior work [52, 54]. Tiles communicate periodically (e.g., every 32 cycles) to find the timestamp of the earliest unfinished task, then commit all earlier tasks. This process leverages large commit queues to commit many tasks at once, achieving commit throughput of multiple tasks per cycle with little communication.

3.2.3 Task unit design

Chronos's task unit consists of two main structures: a *task queue* (TQ) holds all tasks in the tile and dispatches *idle* tasks to PEs, and a *commit queue* (CQ) holds the speculative state of *running* or *finished* tasks, and commits or aborts them as required. In addition, a small *task send buffer* (TSB) receives newly created tasks from PEs and sends them to the right tile. Figure 3-7 details the microarchitecture of each tile and shows these structures, which, together, are similar to a task-level reorder buffer.

Task queue

The task queue consists of two main structures: a *task array* and an *order queue*. The task array is a simple memory that stores the *task descriptor* of every task in the tile. Each task descriptor contains all data needed to run the task: its type, timestamp, object id, and arguments. The order queue holds *idle* tasks and dispatches them to PEs in timestamp order.

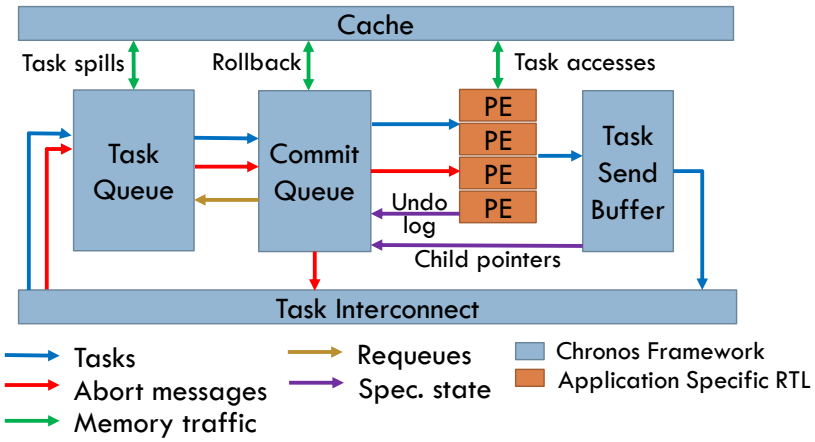


Figure 3-7: Chronos tile microarchitecture.

Tasks allocate entries in the task array and order queue when they arrive to the tile. They hold their order queue entry until they are dispatched to a PE, but hold their task array entry throughout their lifetime (i.e., until they commit or are discarded). This is so that, if the task is aborted, the task array has the information needed to reexecute it. When a task needs to reexecute, it is reinserted into the order queue.

Task spilling: Task queues have limited capacity, but SLOT programs may create an unbounded number of tasks. Chronos provides the illusion of unbounded task queues by spilling tasks to main memory when a task queue is nearly full.

Commit queue

The commit queue holds the speculative state of all tasks that are either running or finished. In Chronos, this speculative state consists of the task's *undo log*, which allows rolling back the task's memory writes, and *child pointers*, which allow aborting the task's descendants.

Each child pointer tracks the tile and task array entry id of a child task. When a child is created, it is sent to the tile specified by its object id. When the receiving tile queues it, it replies with the child task's pointer.

Aborts: Every abort event needs to abort the current task and its data-dependent tasks, and discard all their descendants. Chronos handles all data dependences by aborting the task and all the same-object tasks that come later in program order. Chronos rolls back this sequence of tasks by applying their undo logs in reverse execution order and sending abort notifications to all child tasks, which initiate their own aborts.

Commits: Chronos implements the GVT protocol to perform high-throughput commits, as noted above. Periodically (e.g., every 32 cycles), each tile finds the timestamp of its earliest unfinished task, called the *local virtual time* (this is simply the minimum of the timestamps in the order queue, PEs, and TSB). Tiles send their local virtual times to an arbiter, which finds the minimum, i.e., the timestamp of the earliest unfinished task in the system, called the *global virtual time* (GVT). Finally, the arbiter broadcasts the

GVT. The commit queue in each tile scans its entries in the background, and frees (i.e., commits) those whose timestamp is lower than the GVT.

Deadlock avoidance: Chronos prevents deadlock by *never blocking the lowest-timestamp task*. By induction, this strategy ensures that all tasks ultimately commit. If the lowest-timestamp (i.e, earliest) task cannot be dispatched because the commit queue is full, the commit queue aborts one of its entries (the highest-timestamp entry) and lets the earliest task proceed. To prevent the earliest task from being stalled because the TSB is full, the TSB reserves one of its entries for the earliest task.

Commit entry implementation: Since SLOT tasks are short, the commit queue implements a simple storage format with a fixed number of child pointers and undo log entries. These values are configurable per application. We find that having eight child pointers per task (as in prior work [54]) and eight address-value pairs per undo log suffices for all tasks. Tasks that exceed these limits could be split to use multiple commit queue entries [92]. Alternatively, more sophisticated implementations could support variable entry sizes, or unbounded sizes by spilling to memory [73].

Task send buffer

The task send buffer (TSB) buffers child tasks created by PEs and sends them to their destination tiles. Each entry stays in the buffer until the destination tile acknowledges its receipt. The TSB decouples PEs from task enqueue latency: it lets PEs continue execution and even move to other tasks before the current task's child enqueues have been acknowledged.

Processing Element interface

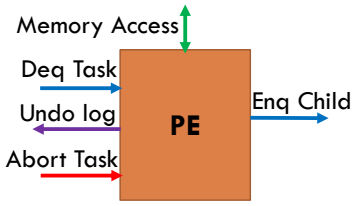


Figure 3-8: PE interface.

The PEs within a tile execute the functionality of each task. If the application program consist of more than one task type, the PEs could either be homogeneous (any PE can execute any task) or heterogeneous (a different PE for each task type). Chronos admits multiple styles of PE, from programmable cores to fully specialized engines, and only requires that they implement a simple interface.

Figure 3-8 details the PE interface, which consists of five ports. All ports use a simple valid/ready handshake mechanism and support pipelining. The PE signals it can accept new tasks and receives them through the *Deq Task* port. It sends children to the TSB through the *Enq Child* port, and issues memory accesses through the *Memory Access*

port. The task unit may abort a running task through the *Abort Task* port. Finally, when the PE finishes or aborts a task, it outputs its undo log through the *Undo Log* port.

Enforcing SLOT restrictions: If a SLOT task tries to access a different read-write object than the task's object id provided at task creation time, the expected behavior is undefined. To ease debugging, Chronos can be configured with a debug aid that detects all memory accesses whose task has an incorrect object id, and stops execution on any such violation.

3.2.4 Chronos customization

Chronos is fully customizable, and allows for three levels of customization.

Customizing processing elements: Chronos allows the application developer to customize the style and functionality of PEs. Chronos only defines the interface, and allows for any implementation that respects this interface. The PE could be simple Finite State Machines (FSM) that implement task logic, or more complicated deeply pipelined engines that support multiple threads in flight. Chronos also supports replacing PEs with programmable cores, and provides RISC-V cores augmented with SLOT ISA extensions so that designers can prototype new algorithms on software before building specialized PEs.

Customizing microarchitecture parameters: Chronos allows customizing parameters such as the size of the task and commit queues, the geometry of the caches (e.g., line size, number of lines, number of ways) and the number of tiles and PEs per tile.

Chronos also allows customizing the width of these queues. This is required because different applications may desire different number of arguments per task. The widths of the inter-tile interconnects are also configured based on the number of arguments per task.

Relaxing operational semantics: Chronos can relax its operational features to take advantage of application characteristics.

For example, some applications can be made resilient to out-of-order writes (e.g., applications that perform directed graph searches, as we will see in Section 3.4). In this case, Chronos can be configured to not perform rollback on aborts. This simplification removes the undo log and commit queue. As we will see in Section 3.4, this saves about 30% of area, enabling designs with more tiles and thus more parallelism.

Even with no-rollback execution, respecting task order is still important. The algorithm may be resilient to order violations, but frequent violations make these algorithms work-inefficient. Therefore, task queues still dispatch tasks speculatively in timestamp order.

Finally, Chronos can also be used for unordered non-speculative applications. In this case, in addition to disabling rollback, the task queue dispatches tasks in FIFO order, removing the order queue and further reducing area.

Task Queue	4096-entry task-array 8192-entry order queue
Commit Queue	128 entries default; 256 for `maxflow`
Task Send Buffer	16 entries
Cache	2 MB/tile default; 1 MB for `sssp`; 4-way set associative, 64B cache lines, 5-cycle hit latency from a PE
Data Widths	32-bit timestamp and object id
Clock Frequency	125 MHz

Table 3.1: Chronos tile configuration parameters used.

3.3 Methodology

Chronos FPGA implementation: We implement the Chronos framework in SystemVerilog. We use a pipelined heap [10] to implement the order queue, and a small CAM adapted from [65] to find conflicting tasks in the commit queue.

Our FPGA implementation is fully configurable in terms of number of tiles, number of cores and their types, and task and commit queue sizes. We use the Amazon AWS FPGA framework [6], and develop Chronos as a CL (Custom Logic) module. This CL module interacts with the AWS Shell, which provides I/O services such as memory and PCI controllers.

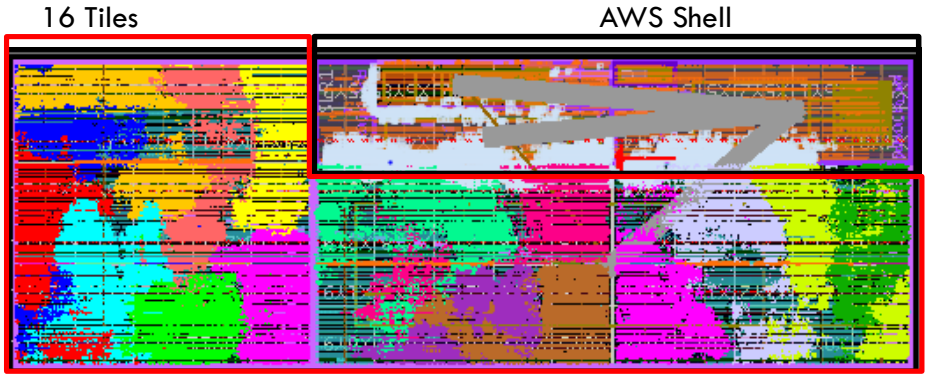


Figure 3-9: FPGA layout of the 16-tile Chronos sssp accelerator. Each Chronos tile is shown in a different color.

We synthesize our CL module using Vivado 2018.1. We target AWS f1.2xlarge instances, which have the Xilinx UltraScale+ VU9P FPGA. This FPGA is fairly large, featuring 1.2M LUTs, 76Mb of Block RAM, and 270Mb of Ultra RAM. We use URAM for the caches and BRAM for the task queues. These resources were sufficient to fit systems of up to 16 tiles while meeting a target frequency of 125 MHz. Table 3.1 details the Chronos configurations used, and Figure 3-9 shows the layout of a 16-tile Chronos system on FPGA.

Application	Baseline	Input	N. Tasks
des	Galois [78]	csa32 [78]	3.1M
maxflow	Galois [78]	rmf-wide [5]	7.8M
sssp	Galois [78]	USA-roads [28]	58.0M
astar	In-house	germany-roads [39],	4.1M
color	[45]	com-youtube [61]	5.8M

Table 3.2: Applications accelerated using Chronos, their baselines, inputs, and total number of tasks executed.

3.3.1 Applications

We build Chronos accelerators for four challenging applications. We compare their performance against highly optimized software-parallel implementations: Table 3.2 details the baselines, input sets used, and the number of tasks executed for these applications.
Baseline system: We run the software baselines on a 20-core/40-thread m4.10xlarge AWS instance. These instances use a 2.4 GHz Intel Xeon E5-2676v3 (Haswell) CPU. We chose this instance specifically because its price is comparable with the FPGA one ($2/hour vs. $1.6/hour for the FPGA instance), which we believe is a fair metric when comparing application performance on different hardware substrates.

Discrete Event Simulation (des)

DES performs gate-level simulation of logic circuits. Our benchmark is similar to the detailed description on Section 2.3.1. The baseline runs the Chandy-Misra-Bryant algorithm (also described previously in Section 2.3.1) implementation in Galois [78]. We do not compare against the software speculative version of the algorithm (TimeWarp) since software speculation adds thousands of cycles per task [17, 46], a huge overhead since each des task takes tens of cycles.

We run this benchmark with the circuit of a 32-bit carry select adder, the largest available in Galois. This is a purely combinational circuit. However, to simulate multiple cycles, and to achieve a longer simulation, we feed it with multiple input vectors at different times.

Maximum flow (maxflow)

maxflow finds the maximum amount of flow that can be pushed from a source to a sink node through a network. This problem models the network as a graph. The edges could either be ordered or unordered, with ordered edges referring to links which allow a one-way flow only. We denote a given node as a source, and assuming an external source can pump sufficient amount of flow into this source, the maximum flow problem computes how much of this flow can come out of the sink.

There are multiple algorithms of computing this maximum flow. We use the push-relabel algorithm [5], which is considered the most efficient with a $O(V^2E)$ complexity.

Push-relabel tags each node with a height. It initially gives heights of 0 to the sink, N (the number of nodes) to the source, and 1 to every other node. Nodes are temporarily allowed to have excess flow, i.e., have more incoming flow than outgoing flow. Nodes with excess flow are considered active and can push this flow to lower-height nodes. The algorithm processes one active node at a time, attempting to push flow to neighbor nodes and potentially making them active. When an active node cannot push its excess flow, it increases its height to the minimum value that allows pushing flow to a neighbor (this is called a relabel). The algorithm processes active nodes in arbitrary order until no active nodes are left.

We use the Galois [78] implementation of the push-relabel algorithm as our baseline. This baseline uses the *inspector-executor* approach to synchronize accesses to multiple nodes. Each transaction first attempts to lock all its neighbors, and proceeds to read and update neighbors' state only if all locks could be acquired. If locking fails, it releases all locks and retries at a later time.

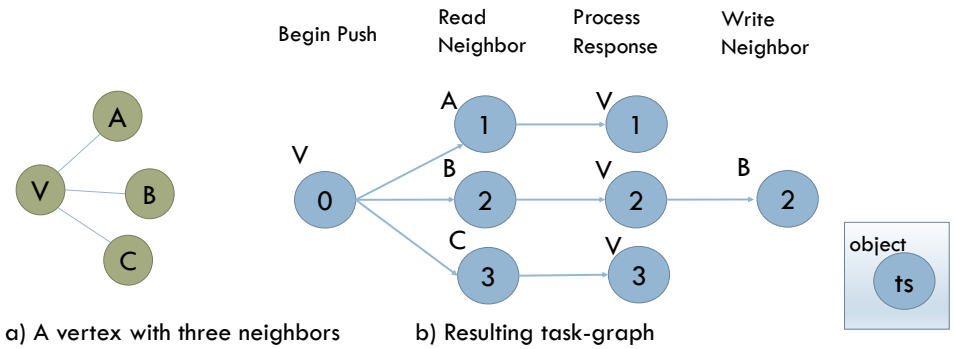


Figure 3-10: Example showing how a coarse-grained maxflow task, which requires atomic access to the neighborhood of graph node, is broken down to multiple tasks that access one object each.

This algorithm is challenging to implement on SLOT because the processing of an active node is a *pull-push* type of operation, where each node reads its neighbors' states, performs some computation, and updates the neighbors' states. Under SLOT, a single task cannot access multiple nodes, so we make use of the procedure outlined in Section 3.1.3 to break this operation down to multiple fine-grained tasks that operate on a single node each. Figure 3-10 illustrates this. Figure 3-10(a) shows a node V with three neighbors. Node V has excess flow (indicated by it receiving a Begin Push task) and needs to push this flow out to its neighbors. It first reads the state of its neighbors to identify which of them have lower height. This is implemented by creating a separate Read Neighbor task for each node, each of which responds with the neighbor's height by creating a separate Process Response task back to node V. If the heights match, then the Process Response task creates a Write Neighbor task to the neighbor's state. This task may also enqueue a new Begin Push task if the neighbor now has excess flow.

As we discussed in Section 3.1.4, this scheme of creating fine-grained tasks has several advantages. First, it facilitates exploiting intra coarse-grained task parallelism;

all neighbors of vertex V are processed in parallel. In addition, if one of the neighbor's tasks, say A, received a conflict, then only the chain of tasks corresponding to A needs to be aborted, not the entire tree.

Single-source shortest path (`sssp`)

`sssp` finds the shortest distance between a given source node and all other nodes in a directed graph. We accelerate the more work-efficient Dijkstra's algorithm which has complexity $O(V + E \log V)$, in comparison to prior accelerators [102] which used the $O(VE)$ Bellman-Ford algorithm.

In Dijkstra's algorithm, we maintain a sequence of tasks. Each task operates on a single node and is ordered by its distance to the source node. `sssp` relies on task order to guarantee that the first task to visit each node comes from a shortest path. This task sets the node's distance and enqueues tasks to visit each neighbor. Later tasks visiting the same node do nothing. Since each task visits a single node, there is no requirement for fine-grained task breakdown as in `maxflow`.

This benchmark admits the no-rollback optimization. Too see why, let's assume that two tasks to the same node gets reordered, say with distances 5 and 10, and the task with distance 10 is run first which sets the node's distance to 10. Next, when the lower-distance task is dispatched, it should first abort task 10. In this case, rolling back its effects involves setting the node's distance to infinity. However, given that after the rollback, task 5 would immediately set the distance back to 5, it is more efficient to skip the rollback on task abort and directly let the aborting task update the distance to a lower value.

We compare `sssp` with the Galois implementation. This implementation uses the delta stepping [71] algorithm. The delta stepping algorithm sorts the tasks into coarse-grained buckets based on their distances, where each bucket spans multiple distances. Then it runs all tasks within a bucket in parallel. The performance of this algorithm is highly sensitive of the size of a bucket, a parameter usually called *delta*. A lower delta value limits parallelism, while a higher delta value increases wasted work in updating nodes to non-final values. We pre-tune delta to the graph used to put the software version in the best possible light.

Astar search (`astar`)

`astar` is similar to `sssp` in that performs a shortest path query. However, `astar` performs a heuristic-directed search towards a goal node, instead of an undirected search as in `sssp`. This brings in two key changes.

First, depending on the heuristic used, each `astar` task performs more compute work than the equivalent `sssp` task. In our case, We evaluate `astar` using a road graph, where the heuristic function is the great-circle distance between two points given their (latitude, longitude) coordinates. Each distance computation requires several trigono-

metric operations, increasing the compute requirements of a task. Specifically, each `astar` task takes two arguments as input, the current distance from source, and the heuristic distance from the node to the goal node. The tasks are ordered by the sum of these distances.

Second, `astar` stops computation as soon as the goal node is reached, compared to `sssp` which does not stop until all nodes are visited. Detecting whether the goal node has been reached is non-trivial in SLOT. Prior software implementations use a global *done* variable that is set when the goal node is reached. Each task exits early if this done variable is set. However, implementing a done variable is difficult in SLOT, since it needs to be treated as a separate read-write object, and each task can only access a single read-write object.

Hence, we extend Chronos to support this special semantic. When the goal node is reached, it enqueues a special *termination task*. After a termination task commits, Chronos discard all future tasks, ending the computation early.

Like `sssp`, `astar` admits the no-rollback optimization, so our evaluation uses this variant. We could not find a high-performance parallel implementation of `astar`. Therefore, we implement our own, using Galois's delta stepping algorithm, and use it as the baseline.

Graph coloring (`color`)

We also implement a non-speculative graph coloring algorithm to show that Chronos can be used with non-speculative tasks. `color` assigns a color to all graph nodes such that no two adjacent nodes share the same color. We use the Jones-Plassmann [56] algorithm with the largest-degree-first heuristic. This algorithm sorts nodes by their degree. Each round, the highest degree node is selected, and it is assigned the least-indexed color that none of its neighbors have already been assigned with.

In our non-speculative implementation, we use a push-based approach since it naturally maps to SLOT. We initially assign the color 0 to all nodes with no higher-degree neighbors. After assigning a color to a node, it sends a task to all its lower-degree neighbors informing them of the choice of the color. Each node maintains a count of higher-degree neighbors with with a bitvector of colors that the neighbors have already been assigned with. Once a node receives messages from all its higher-degree neighbors, it assigns a color to itself and inform its lower-degree neighbors. This process continues until all nodes have been colored.

We compare against a baseline implementation from Hasenplaugh et al. [45]. This implementation implements the same algorithm as above, but instead uses a pull-based approach, where each node queries its neighbors for their chosen color. This pull-based approach makes sense for shared memory systems, but not for SLOT.

All `color` tasks are unordered, and rely on object ids to serialize tasks for the same node.

3.3.2 PE implementations

Specialized PEs: We write specialized PEs for each application in SystemVerilog. Our PEs are pipelined, with 4-30 pipeline stages per PE. Each stage performs some computation and may issue a single memory access. Pipelining is flexible: tasks stalled on a memory access do not block other tasks, which can overtake them and proceed to later stages. Each PE has sufficient storage for 32 in-flight tasks. However, we find that a single PE often saturates task and cache bandwidth with fewer in-flight tasks.

PEs can also be generated from a C-like description using High-Level Synthesis (HLS). Specifically, `astar`, which has trigonometric computations that are tedious to write in SystemVerilog, uses HLS to generate most of the pipeline.

RISC-V cores: In addition to application-specific cores, we also built a Chronos version with RISC-V cores. We use the Spinal HDL RISC-V core generator [90] to generate a 32-bit core that has a performance of 1.2 DMIPS/MHz. We extend these cores to implement Chronos `enqueue` and `dequeue` operations, and use the interrupt logic to abort running tasks.

We write SLOT C implementations of `des`, `maxflow`, `sssp` and `color`, compile them to the RISC-V ISA, and run them on a 48-core Chronos system (4 tiles with 12 cores each). We later compare this system with those using application-specific cores to quantify the benefits of PE specialization.

For all timing measurements, we ignore the benchmark setup time, including data transfer time between the host CPU and FPGA, and only consider the runtime of the algorithm. This is reasonable since the time spent transferring data can be amortized over multiple queries on the same graph. We instrument the FPGA design to count cycles, profile efficiency, and measure detailed component utilization.

3.4 Evaluation

We first compare the performance and scalability of the Chronos FPGA accelerators with application-specific PEs against the software-parallel versions on the Xeon CPU.

Figure 3-11 shows the speedup of each of the four applications as the number of threads or tiles grows until they fill each system. Because each design has different numbers of tiles, the x-axis is percentage of system utilization. For the software-parallel versions, we sweep the number of threads from 1 to 40. For FPGA versions, 100% system utilization corresponds for the maximum system size we can fit on the FPGA (6-16 tiles, depending on the application). Table 3.3 reports this maximum size. To obtain results at lower utilization, we first disable tiles, and then disable the number of concurrent tasks on a single-tiled system. `sssp` and `astar` admit the no-rollback optimization. Table 3.3 also summarizes the performance of the FPGA and baseline implementations at the best-performing system sizes.

For all four ordered applications, Chronos performs better than the best CPU base-

App	Tiles	FPGA vs serial CPU			CPU	Overall
		1 task[1]	1 tile	all tiles	scaling	speedup
des	8	2.45×	26.8×	109.9×	7.2×	15.3×
maxflow	8	0.11×	0.7×	4.3×	1.0×	4.3×
sssp	16	0.24×	3.8×	48.4×	13.3×	3.6×
astar	6	0.58×	16.9×	74.4×	21.2×	3.5×

[1]A single concurrent task running on a single PE.

Table 3.3: Performance and scalability of Chronos accelerators with specialized PEs vs. the best CPU performance.

line, achieving a gmean speedup of 5.4×. Chronos is 15.3× faster than the CPU version on `des`.

Table 3.3 also shows the progression of speedups relative to a serial CPU implementation as the FPGA system is scaled from a single concurrent task, a single tile, and the full system.

Note that the FPGA runs at a 19× slower frequency than the CPU. Hence, except in `des`, the FPGA version with a single concurrent task is substantially slower than the serial CPU version. This slowdown is typically 1.7–9×, better than the 19× difference in frequencies, because the FPGA PEs are customized to each application.

Nonetheless, Chronos more than makes up for this handicap by *scaling to many concurrent tasks*: Chronos accelerators all scale well, sometimes beyond 100×, because they exploit abundant parallelism that is not available to CPU versions.

3.4.1 Application analysis

We now look at each application in detail.

des: Even when running a single task at a time on one PE, Chronos is actually 2.45× faster than the baseline (left column of Table 3.3). This happens even though Chronos is running at a 19× lower frequency. This is primarily because the baseline maintains a priority queue in software, whereas the Chronos framework provides a much higher throughput implementation in hardware. As the number of concurrent tasks increases, the Chronos implementation scales to a self-relative speedup of 44.9× at 8 tiles, corresponding to a 15.3× speedup over the best CPU implementation.

maxflow: The Chronos instance that runs a single task at a time is 9× slower than CPU (2.1× faster after accounting for the frequency difference), mainly because the baseline `maxflow` does not use priority queues and hence is not as hampered as in `des`. However, in the baseline implementation, the tasks are large and therefore parallelism is scarce, achieving a maximum speedup of 3% at 6 threads, before crashing down to a 3.7× slowdown at 40 threads, due to overwhelming synchronization costs at higher thread counts.

The Chronos implementation uses more fine grained tasks, which uncovers huge

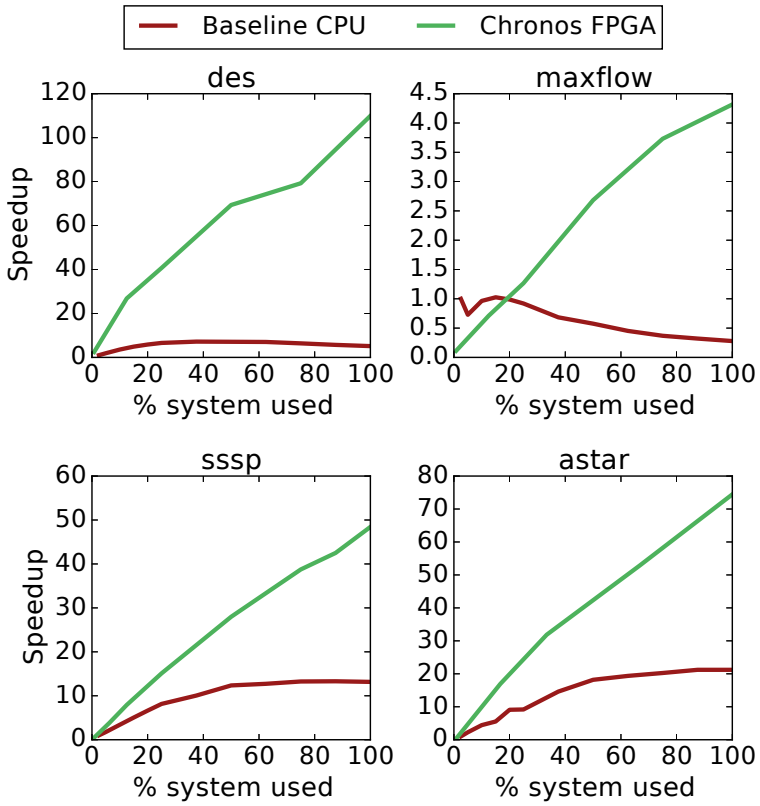


Figure 3-11: Scalability of Baseline CPU and Chronos implementations of the ordered algorithms. For CPU, 40 threads = 100% system. For Chronos, 100% corresponds to the number of tiles in Table 3.3. Speedups are normalized to the serial CPU version.

parallelism, with a self-relative scalability of 39.9×, resulting in an absolute speedup of 4.3×.

`maxflow` thus shows the benefits of dividing large unordered transactions into small ordered tasks, as Section 3.1.4 outlined.

sssp: Similar to `des`, baseline `sssp` also uses a priority queue to schedule tasks, which Chronos provides in hardware. Hence, performance with a single concurrent task is 4.5× larger than the frequency-adjusted 0.05×. But unlike `des`, `sssp` tasks are resilient to order violations. The baseline uses this insight to obtain a 13.3× speedup at 40 threads. However, Chronos scales even further, up to a 202× self-relative speedup, to give a 3.6× performance advantage over the baseline.

astar: `astar` follows a similar pattern to `sssp`. The performance with a single concurrent task is 1.7× slower than the CPU, and both CPU and FPGA versions scale near linearly. But the FPGA can fit significantly more concurrent tasks than the CPU, achieving a speedup of 3.5×.

In conclusion, Chronos FPGA accelerators uncover significantly more parallelism than their baseline CPU implementations, enough to consistently outperform the CPU despite their much lower frequency. Thus, we expect that high-frequency ASIC versions would achieve even higher speedups.

Comparison with Swarm: In Chapter 4, we extend Chronos to support the Swarm execution model, and present a comparison with Swarm in Section 4.4.

3.4.2 Chronos on non-speculative algorithms

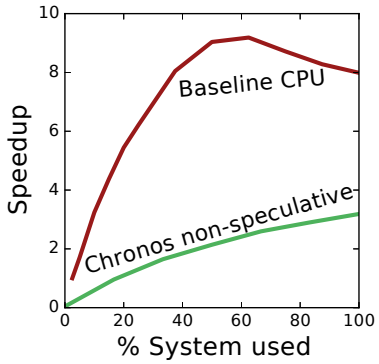


Figure 3-12: Scalability of unordered `color`.

Figure 3-12 shows the scalability of non-speculative `color`. Chronos achieves a self-relative scalability of 45×. But the baseline also scales to 9.1×. Though Chronos uncovers more parallelism, it is not sufficient to make up for the 19× penalty in frequency, so the FPGA version is 2.9× slower than the CPU overall. This result shows that Chronos is not necessarily profitable when the software version has easy parallelism (i.e., simple synchronization and sufficient scalability).

3.4.3 Analysis of system efficiency

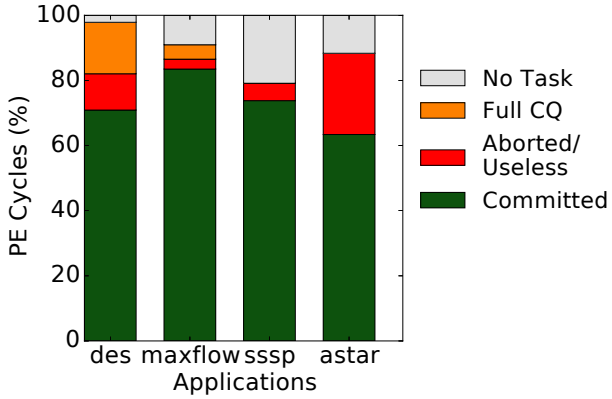


Figure 3-13: Breakdown of aggregate PE cycles in speculative Chronos variants.

To analyze the efficiency of speculative execution, we look at how each PE spends its cycles for the four speculative applications. Figure 3-13 breaks down PE cycles into those spent running *(i)* tasks that are ultimately committed or *(ii)* tasks that later aborted or, for the no-rollback applications, tasks that performed useless work; and cycles where the PE was stalled due to *(iii)* a full commit queue or *(iv)* the task queue not having any task to dispatch.

Figure 3-13 reveals two insights. First Chronos spends most cycles on tasks that ultimately commit. Only 11% of cycles are spent on aborted or useless work overall. Second, the commit queue size, i.e., the number of tasks that can be speculated ahead, moderately limits performance on applications without the no-rollback optimization.

Impact of the no-rollback optimization: We have also generated Chronos accelerators for `sssp` and `astar` without the no-rollback optimization. Due to the higher area requirement of enforcing rollback (Section 3.2.4), we were only able to fit 8 tiles for `sssp` (compared to 16). As a result, the performance of with-rollback versions are 2.3× slower for `sssp` and 4.1× slower for `astar`. The slowdown for `astar` is because `astar` with rollback suffers from large commit queue stalls.

Queue utilization: Figure 3-14 shows the average number of task and commit queue entries used across the system by each speculative application. Each tile has a 4K-entry task queue, with the number of tiles specified in Table 3.3. Large task queues are important for `sssp` and `astar`, which use more than 4K entries on average.

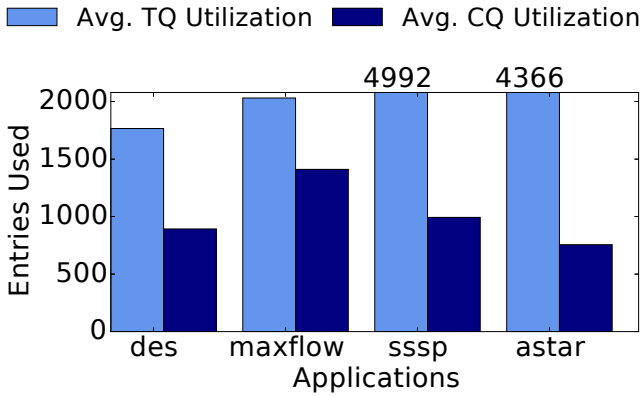


Figure 3-14: Task queue and commit queue utilization (over all tiles) of each application.

Figure 3-14 also shows the commit queue utilization. Since `sssp` and `astar` no-rollback versions do not use the commit queue, this graph shows the results for versions with rollback. All applications use 700–1500 commit queue slots across all tiles on average, showing that applications need a large window of speculation to uncover enough parallelism.

Benefits of specialization: Figure 3-15 quantifies the benefits of using application-specific PEs over general-purpose RISC-V soft-cores. Overall, these results show that application-specific PEs have a 4–11× performance advantage.

Estimated ASIC performance: Finally, we use our FPGA prototype to evaluate the benefits of an ASIC Chronos implementation. We estimate that an ASIC RISC-V Chronos implementation could run at 2 GHz, a 16× higher frequency than the FPGA prototype's 125 MHz. To emulate this higher frequency, we throttle DDR memory bandwidth by 1/16th, since off-chip bandwidth would not change with frequency.

We find all applications except `color` are not bandwidth-bound and the 2 GHz ASIC achieves a 16× performance improvement over the 125 MHz FPGA (the FPGA prototype

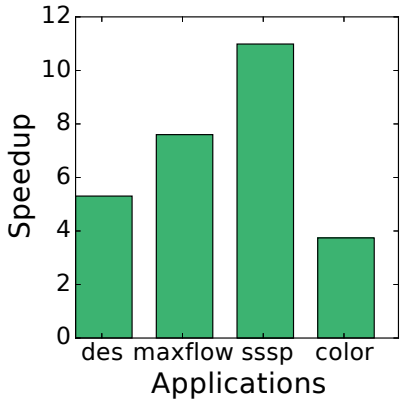


Figure 3-15: Performance advantage of using application-specific PEs over RISC-V soft-cores.

	Available	TQ	CQ	TSB	Cache	des	maxflow	sssp	astar	color
LUTs (K)	895	17	12	0.5	12	7	11	4	10	7
FFs (K)	1790	6	8	0.3	12	7	6	4	10	8
BRAM	1680	38	5	-	72	-	-	-	-	-
URAM	800	-	-	-	64	-	-	-	-	-

Table 3.4: Per-tile FPGA resource consumption for each of the framework components and application-specific PEs

has a memory bandwidth of about 50 GB/s). For `color`, the improvement is limited to 13.7×. Thus, compared to the CPU baseline, an ASIC RISC-V Chronos would achieve speedups ranging from 4.7× (`color`) to 244.8× (`des`). Compared to having specialized PEs on an FPGA, speedups would range from 1.5× (`sssp`) to 3.7× (`color`).

3.4.4 Analysis of implementation costs

Lines of code: Chronos makes it simple to design custom accelerators to extract speculative parallelism. The Chronos framework components take over 20000 lines of SystemVerilog. By contrast, each application is much simpler: `sssp` takes just 100 lines, `des`, `maxflow`, and `color` around 300 lines, and `astar` is around 600 lines.

FPGA utilization: Table 3.4 shows the FPGA resource consumption of each framework component and PE. Overall, we observe that, while the framework components consume substantial resources, they are comparable to those of PEs, which are very simple.

3.5 Conclusion

We have presented Chronos, the first framework to build accelerators for applications with ordered speculative parallelism. Chronos makes speculative execution cheap by relying on SLOT, a new execution model that limits tasks to access a single read-write object, avoiding the need for cache coherence.

We implement Chronos on an FPGA and use it to accelerate several challenging applications in graph analytics and simulation. We deploy these accelerators on commodity AWS FPGAs, where we demonstrate 5.4× gmean speedup for the same applications over their software-parallel versions.

CHAPTER 4

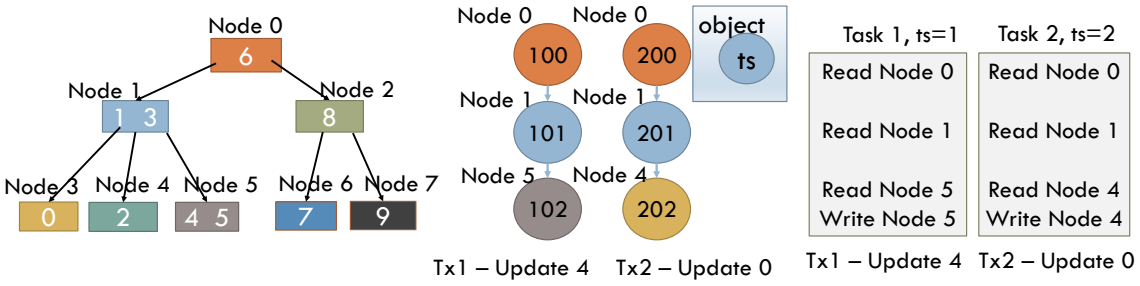
Extending Chronos to Support the Swarm Execution Model

In the previous chapter, we have seen that SLOT achieves excellent performance on a number of applications. In this chapter, we demonstrate some of SLOT's limitations and show how the Swarm execution model, already described in Section 2.3.3, can mitigate these limitations. The Swarm execution model is similar to SLOT, where the program is expressed as a set of tasks ordered by timestamps. But unlike SLOT, which restricts tasks to accessing a single read-write object, Swarm allows a task to access arbitrary data.

In Section 4.1, we will show how loosening SLOT's single read-write restriction can benefit some applications, via an example of B-tree traversal. We then present our Chronos-Swarm extension, which adds new hardware to Chronos to keep track of all accesses made by a task and find and abort conflicting tasks (i.e., higher-timestamp tasks that accessed the same data with at least of the accesses being a write) across tiles.

We use Chronos-Swarm to implement an accelerator for `ycsb`, a transactional database application. We compare its performance with an accelerator for `yscb` generated from Chronos-SLOT and find that Chronos-Swarm implementation is 3.5x faster, with an area overhead of 10%.

We also generate Chronos-Swarm accelerators for the three other applications evaluated with SLOT. However, for these application, Chronos-Swarm is slower by up to 2×. We discuss the reasons for these differences, and use it to motivate the hybrid SLOT/Swarm execution model that we will present in Chapter 5.



(a) Example B-tree. Each node contains a set of keys.

(b) SLOT implementation of two update operations.

(c) Swarm implementation of two update operations.

Figure 4-1: Example B-Tree application, and its SLOT and Swarm tasks

4.1 Motivation

SLOT is a data-centric execution model. Each object has a defined home tile, and any compute on that data must be run near that home. However, there can be situations when an alternative compute-centric model, where data is replicated across tiles is more efficient. We first illustrate these tradeoffs with a concrete example.

4.1.1 Transaction processing on a B-tree

Consider a key-value store application, where the pairs are stored stored in a B-tree structure (Figure 4-1(a)) [26]. A B-tree is an n-ary tree. Each tree node contains n key-value pairs and n+1 child pointers. B-tree maintains a sorted order of keys as follows: Within a node, keys are in sorted order, and each key falls between two child pointers. For each key, k, in a parent node, each key in the left child is less than k and each key in the right child is greater than k. Here, we assume the keys are unique.

Suppose our key-value store implements transactional operations, where a transaction may read or modify multiple key-value pairs. These operations require searching for a given key by performing a traversal of the tree and potentially updating its associated data value. For this example we will assume that both keys and values are integers.

Each transaction accesses multiple tree nodes whose contents may be modified. So a SLOT program should isolate each node visit in its own task. Figure 4-1(b) shows how we can break each transaction into SLOT tasks. Each transaction is assigned a global timestamp range, e.g., range 100-199 for Tx1. Each transaction begins with a SLOT task for the root node. This task compares the search-key with the keys already present in the node. If the search-key is not found, it enqueues a new task for the appropriate child. Once a task has found the search-key, its associated value is updated and the transaction ends.

This B-tree traversal example shows several of SLOT's drawbacks.

First, all tasks within a transaction are sequential. This division exposes no more

parallelism than a single task. But keeping track of multiple tasks consume more on-chip resources, such as commit queue entries. When these entries fill up, it limits the speculation window. For example, if a tile receives a task with a lower timestamp than any task in the commit queue, it must be abort one task to make space in the commit queue. This situation is highly likely in a B-tree traversal because the earliest ordered transaction keeps creating new tasks while the tasks for later-ordered transactions are waiting in the commit queue.

Second, all transactions start with a SLOT task at the root node. All of these root tasks would be mapped to the same Chronos tile, which needlessly serializes these tasks and may also lead to load imbalance.

However, we note that most tasks that access the root are read-only. A task would only update the root node when an insertion or deletion propagates all the way up to the root, a rare occurrence on a multi-level tree. A shared-memory implementation (like Swarm) would allow the root (and other rarely written top-level nodes) to be replicated across all tiles, mitigating the load imbalance issue. However, a shared-memory implementation is not ideal for lower-level nodes. Lower-level nodes will have frequent writes, which cause invalidations. Therefore, a partitioning strategy where a single node is only allowed to be accessed from a single tile would perform better.

4.1.2 Replicating vs. partitioning

The above example shows that depending on the frequency and type of use, some data may be better replicated across caches, while others may be better served with a partitioned strategy.

To understand when each strategy is better, we consider two parameters of a data item, *write intensity* and *stack distance*.

Write intensity is the ratio of reads to writes. Read-only data have a write-intensity of zero, while a value of one indicates each write is used once. For data items with low write intensity, such as the top-level nodes of the B-tree, it is preferable for them to be replicated across the chip reducing read latency. Figure 4-1(c) shows how the B-tree traversal is mapped assuming the entire graph is allowed to be replicated.

Alternatively, a partitioned model is preferable for data with high write intensity, such as the leaf nodes of a B-tree. This would ensure that all writes happen from one place and save the coherence traffic taken up by invalidation messages.

There can be other situations where even data with low write intensity may be better suited to a partitioning strategy. For instance, consider a middle layer of the B-tree. Nodes in this layer may still be rarely written. But these nodes may not be read that frequently either. Replicating these nodes may waste cache capacity, so a partitioning strategy may be desirable.

We formalize this notion through the concept of stack distance [70]. The stack distance of an access A is defined as the number of distinct data items accessed between A and a prior access to the same data item as accessed by A. For example, the stack

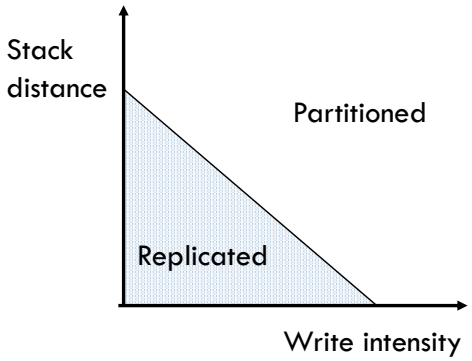


Figure 4-2: Reuse distance and write intesity affect whether a data-centric or compute-centric model is preferable

distance of the second access to *b* in a trace "*b a c c b*" is two because two distinct data elements *a* and *c* are accessed between the two accesses to *b*. Data with a higher stack distance may be better served partitioned as it prevents the same data unnecessarily being replicated on multiple caches.

Figure 4-2 summarizes these concepts. Data with high write intensity or high stack distance are better accessed under a partitioned model, while data with low write intensity and low stack distance are better served with a replicated model.

In summary, SLOT forces all data to be served with a partitioned strategy. This suboptimal for data items with low write intensity and low stack distance. Therefore, we extend Chronos with another execution model that also supports replicating data across caches, exploiting both strategies.

4.2 Chronos-Swarm

We extend Chronos to support the Swarm execution model, discussed in detail on Section 2.3.3. Swarm is similar to SLOT in that it consists of timestamped tasks. However, unlike SLOT, Swarm tasks do not have an object id and does not impose any restrictions on data a task may access.

Under Swarm, the B-tree benchmark becomes drastically simplified. Each transaction can now be performed in a single Swarm task. Figure 4-1(c) illustrates this.

We call our implementation Chronos-Swarm. Since Swarm allows a task to access any data, the conflict detection protocol in Chronos-Swarm differs from that of Chronos-SLOT. Specifically, Chronos-Swarm needs to ensure that, on each memory access, no other higher-timestamp task in the system has issued a conflicting access to the same line, where two accesses are deemed to conflict if at least one of them is a write.

We first discuss the changes necessary to support this requirement in a single-tile context, and then discuss extensions to multiple tiles.

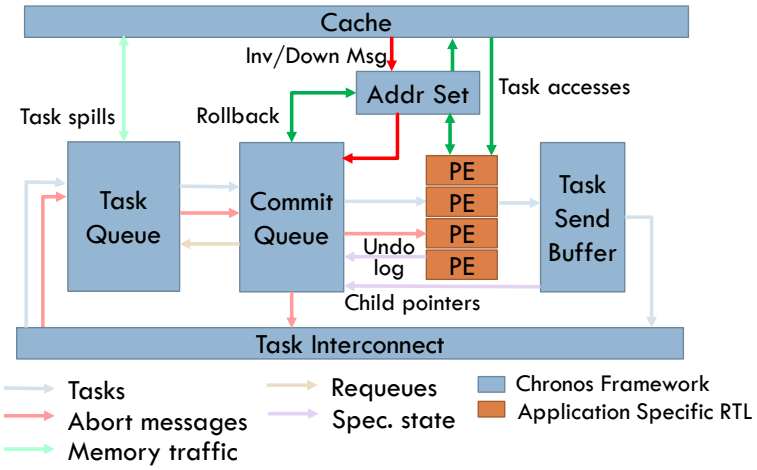


Figure 4-3: Chronos tile microarchitecture for Swarm execution model. The new Addr Set module stores read/write sets of all tasks and performs intra-tile conflict detection on memory accesses.

4.2.1 Chronos-Swarm with a single tile

The primary difference between Chronos-SLOT and Chronos-Swarm is when conflict detection takes place. In SLOT, conflicts are detected based on the object id, which is known before the task starts. Therefore Chronos-SLOT performs conflict detection before a task starts. In contrast, Chronos-Swarm does not detect any conflicts before execution. All conflicts are detected on memory accesses.

Figure 4-3 outlines how a Chronos tile is reorganized for supporting Swarm. The main addition is the Address Set block, which stores the read- and write-sets of each task (Section 4.2.3 describes the implementation of this module in detail). All new memory accesses from PEs are conflict-checked in the Address Set. Read requests are checked against the write-set of all other tasks, and write requests are checked against both read- and write-sets. If any conflicting tasks are found (i.e., higher-timestamped tasks that accessed the same line), the commit queue is instructed to abort them. While the conflicts are aborted, the original request is stalled through a negative acknowledgement (NACK). The PE will retry NACKed requests at a later time.

Similar to Chronos-SLOT, Chronos-Swarm should roll back the effect of any writes that aborting tasks may have already made. These rollback writes are treated similarly to normal writes. They perform conflict checks, and can cause other tasks to abort [54].

4.2.2 Chronos-Swarm with multiple tiles

Chronos-SLOT maps tasks with the same object id to the same tile, ensuring that no conflicting access to the same data will be made across tiles. However, Chronos-Swarm does not do that. A given memory address may be accessed from all tiles in the system.

A naive implementation of multi-tile conflict detection would be to broadcast each memory access to all tiles and wait for them to abort higher-timestamp conflicting tasks.

However, this would be expensive, so we rely on the coherence protocol to filter out most inter-tile conflict checks.

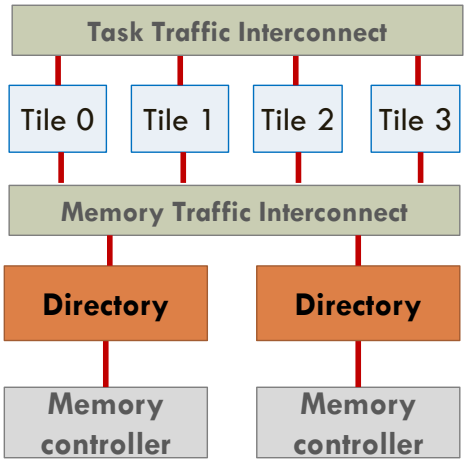


Figure 4-4: Chronos-Swarm high-level view.

Conflict detection by extending a coherence protocol

We extend Chronos with a standard coherence protocol. To keep track of coherence state, we add a coherence directory. This directory sits outside of the tiles, but closer to the memory controller Figure 4-4.

The coherence protocol is a standard invalidation-based MSI protocol. Invalidation and downgrade messages received by a tile cache are used to perform conflict detection. Figure 4-5 illustrates this process.

Figure 4-5 shows a two-tiled Chronos-Swarm system. Tile 0 has one task with a timestamp of 10, while Tile-1 has a task with timestamp 20. Both tasks access line X. Initially, task 20 has run, and Tile-1's cache has the line in M(odified) state. In (a), task 10 performs a write, which causes a GetX (Get exclusive) request to be sent to the

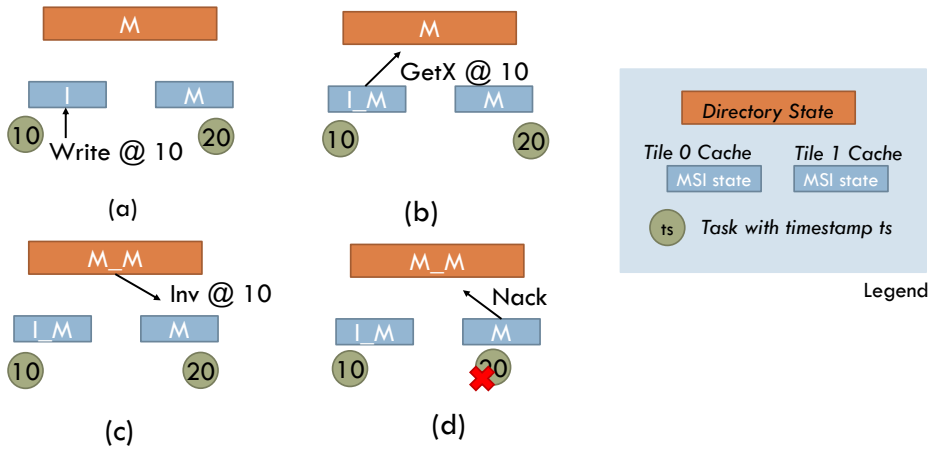


Figure 4-5: Chronos-Swarm uses the coherence protocol to detect conflicts

directory. However, before this request can be served, the directory needs to invalidate the copy in Tile-1. When Tile-1 receives the invalidation request, it aborts task 20, and NACKs the invalidation request. The NACK would propagate all the way to Tile-0, which would retry the request after a time interval.

Here, because the invalidation was NACKed, Tile-1 keeps the line in M state. This has an additional benefit. When task 20 aborts, it must perform a rollback write to restore the data in line X to what it was before task 20 modified it. This rollback write follows the same process as a normal write. Keeping the line in M state allows this write to be completed without any additional coherence traffic.

Sticky Sharers: However, simply relying on standard invalidations may cause some conflicts to be missed. For instance, if in Figure 4-5, Tile-1 evicted the line before the Write @ 10, the directory will remove its record of Tile-1 being a sharer, and will not send any invalidations. However, task-20 still conflicts with task 10 and needs to be aborted.

We solve this issue by using *sticky* sharers, where the sharer-bits in the directory are only cleared when there are no uncommitted tasks that accessed the line [54, 73]. This clearing is done lazily, as part of invalidation and downgrade acknowledgement messages.

Sticky sharers need to be preserved across directory evictions. Hence, these sticky sharer information is backed in the main memory. Whenever a line is evicted, sticky sharers need to be written back, and whenever a new line is brought in from the main memory, an additional memory access must be performed to bring in the sticky sharers as well. As we will see in Section 4.4.3, this additional main memory traffic consumes significant bandwidth.

Canary timestamps: However, sticky sharers are not sufficient to detect all conflicts. Let us consider the same example, but now imagine the tasks happened in the reverse order. Task 10 happened first in Tile-0 and when task-20 ran in Tile-1, it read the updated value written by task-10. Tile-1 now has the line in M state. At this point, if a new task with timestamp 5 starts in Tile-1, This task reads X; this read should abort task 10, but since it hits in Tile-1's cache, an invalidation message will not be sent to Tile-0, and as a result task 10 will not be aborted.

We resolve this situation by augmenting each cache line in the private caches with a *canary timestamp*. When a new line is installed in the cache, the installer's timestamp is set as the line's canary. When a later access comes through, if the requester's timestamp is less than the canary, we deem the access a miss, and sends a request to directory to refetch it. In the above example, task 20 first installed the line X in Tile-1's cache, so this line will have a canary of 20. When the request from task 5 comes in, there will be a canary violation and the line would be refetched. The directory will now see Tile-0 as a sticky sharer and send an invalidation, causing task-10 to be aborted.

Verification: Conventional cache coherence protocols are often verified using model checking tools like Murphi [31]. These tools exhaustively search the state space to

ensure that coherence invariants are not violated. However, this approach does not scale to Chronos-Swarm. The coherence protocol is very tightly coupled to the conflict detection and resolution mechanisms, and is difficult to verify independently. Joint verification will suffer from state explosion and likely intractable.

To increase our confidence in the correctness of our implementation, we create a custom simulator. This simulator performs random searches instead of exhaustive searches, through the state space. We model the functionality of each of the components in Figure 4-3, connected by message queues. We initialize the simulator with a static list of tasks. Each task has a distinct timestamp and performs one read or write to a single address. The simulator executes the tasks in a random order, allowing any message ordering that can happen in Chronos. At the end of a single run the simulator checks that each accesses read the value of the last write in timestamp order (the coherence invariant). This approach is similar to a litmus-test approach used in verifying whether a given implementation correctly implements the defined memory consistency model [68].

We have used this simulator on systems with up to 10 tiles and 10 tasks. We ran each simulation 10M times with different random seeds without encountering any invariant violations. Therefore, we deem that sticky sharers and canary timestamps are sufficient for detecting all conflicts across tiles.

This simulator checks for preservation of coherence invariants, but not forward progress. To ensure forward progress, Chronos-Swarm uses an exponential backoff algorithm. After a NACK, a request is retried with a random delay, where the bounds of the random delay increase exponentially. Specifically, after the N^{th} NACK, the access is retried after rand(0, 2^N) cycles.

Chronos-Swarm task mapping strategy

Chronos-SLOT uses object id to map tasks across tiles. Since Swarm does not have a concept of object id, Chronos-Swarm uses a random task mapping strategy. We also investigate a hint-based strategy [53], where each task is tagged with a hint (this is similar to an object id but is optional), which is then used to map tasks across tiles.

When using hint-based mapping, Chronos-Swarm can be configured to serialize tasks with the same hint. Note that with Chronos-SLOT, serializing same-object tasks was required for correctness, but with Chronos-Swarm, serializing same-object tasks is optional. Tasks with the same hint are likely to conflict with each other, and therefore serializing same-object tasks has been shown to improve performance [53]. However, if the tasks are read-only (e.g., root node tasks in B-tree traversal), then serializing hurts performance. Therefore, we leave the choice of whether to serialize or not as a configurable parameter.

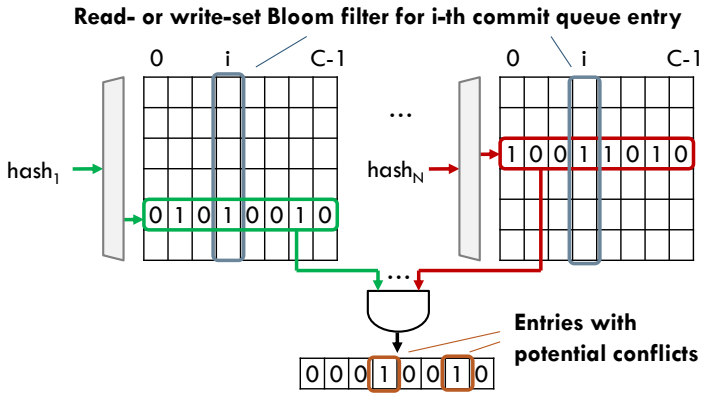


Figure 4-6: Bloom filters store the read/write-sets in columns. A single access reads a bit from all ways, and AND'ing them provides a vector of potential conflicting tasks.

4.2.3 Address Set microarchitecture

We now look at the implementation of the Address Set module.

A task could access a potentially unbounded amount of memory addresses. But we need to store its read- and write- sets with bounded resources. To achieve this, the Address Set uses Bloom filters [12]. We use separate Bloom filters to store the read- and write- sets.

A Bloom filter consists of k n-bit vectors. Recording a new access involves using k different hash functions to select one bit each from all k vectors, and setting those bits. Detecting conflicts requires checking if an address is already in the set. This can be done easily, by hashing the address again, and if the corresponding bits on all k vectors are set, we deem there is a potential conflict. Note that Bloom filters allow for false positives, where a conflict is detected even when there is none.

The above process describes how to detect conflicts with a single task. However, the Address Set module needs to perform parallel conflict checks with all tasks currently in the commit queue. An efficient way of doing this is to store the vectors of all tasks in an SRAM. If we store a single task's vector in a column of this SRAM, a single row-read would provide a mask of all potential conflicts. Figure 4-6 illustrates this process. After obtaining this vector, the Address Set module needs to detect whether these are actual conflicts, i.e., whether the requesting task has an earlier timestamp than the potentially conflicting task. If so, the conflict is real and the conflicting task is aborted. The Address Set module performs this timestamp check with all potentially conflicting tasks sequentially.

When a task leaves the commit queue, after it committed or aborted, we need a way of recycling the entry. This involves clearing all bits in a column. Swarm's proposed implementation [54] uses SRAM banks with perpendicular ports to support row-wise access for reads and column-wise access for writes. However, such structures are rarely available on standard cell libraries and definitely not available on FPGAs. Therefore, Chronos uses a different approach.

Task Queue	4096-entry task-array 8192-entry order queue
Commit Queue	64 entries
Task Send Buffer	16 entries
Cache	2 MB/tile default; 4-way set associative, 64B cache lines
Data Widths	32-bit timestamp and object id
Clock Frequency	125 MHz
Memory	4 DDR Memory controllers
Directory	(only in Chronos-Swarm) 4 banks, each bank 2MB, 4-way set associative
Bloom Filters	128-bit, 4-way bloom filters, H3 hash functions [18]

Table 4.1: Configuration parameters used to compare Chronos-SLOT and Chronos-Swarm.

Chronos amortizes the cost of clearing bits across multiple columns. A background process sequentially traverses all rows of the table and clears the bits of those entries that are no longer valid. If the commit queue slot becomes recycled before this background process completely clears all bits in a column, this would increase the probability of a false positive, but does not impact the correctness. To decrease the probability of such false positives, we allow an entry to be cleared until the new task makes its first memory access, allowing some additional time for the filter entry to get cleared.

4.3 Methodology

We implement Chronos-Swarm in SystemVerilog, and use it to generate FPGA accelerators.

We use the same applications as in Section 3.3 and add one more application, `ycsb`, a transactional database application which uses a B-tree to store key-value pairs. We intend to compare the different tradeoffs when using SLOT vs Swarm execution models. To this end, we generate Chronos-SLOT and Chronos-Swarm accelerators using the same parameters, detailed in Table 4.1. Note that, for the Chronos-SLOT applications that supported no-rollback optimization, this evaluation uses the variant with rollback.

For `yscb`, we initialize the B-tree with 256K random pairs. This size may seem small as a B-tree of this size fits mostly on-chip. However, a larger tree only yields a few extra levels in the tree, and limited extra compute per transaction. Therefore, using a larger tree should not result in major changes in the relative performance between accelerators generated from Chronos-SLOT and Chronos-Swarm.

Each B-tree node contains between 2-4 pairs. We pre-generate 1M transactions with a 50/50 split on reads vs. writes. Each transaction traverses the B-tree to find the tuple with a given key, and potentially update its value.

4.4 Evaluation

We first evaluate `ycsb` on Chronos-SLOT and Chronos-Swarm and then evaluate the rest of the applications from the Chronos-SLOT suite. Finally, we compare the resource consumption and other overheads of the two systems.

4.4.1 Performance of ycsb

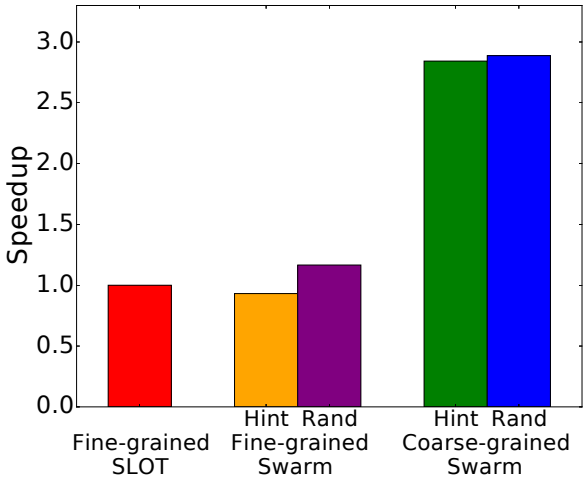


Figure 4-7: Performance of `ycsb` under Chronos-SLOT and different variants of Chronos-Swarm (RISC-V cores).

Figure 4-7 shows the performance of `ycsb` under different schemes with RISC-V cores. We have two variants of `ycsb`; the fine-grained `ycsb-fg` variant in Figure 4-1(b) and the coarse-grained `ycsb-cg` variant in Figure 4-1(c). Note that `ycsb-fg` can be run using both the SLOT and Swarm execution models, while `ycsb-cg` requires Swarm. For Chronos-Swarm, we present results with both hint and random task mapping strategies. For the hint strategy, we use the node id as hint for `ycsb-fg` and the search-key as hint for `ycsb-cg`. All results are normalized to Chronos-SLOT.

The Chronos-SLOT implementation is very similar to Chronos-Swarm's `ycsb-fg` with hints. Both use the same task structure with same task mapping strategy. However, the latter is about 7% slower. This difference arises because Chronos-Swarm detects conflicts later, while the task is running, while Chronos-SLOT detects them before the task is dispatched. This results in tasks running for longer before being aborted. Other than increasing wasted work, this also has the indirect effect of taking up valuable commit queue space and limiting the window of speculation.

Using random task mapping improves `ycsb-fg`'s performance by 25%. This is because random task mapping allows the tasks accessing top-level nodes to be distributed around the system, decreasing load imbalance.

`ycsb-cg` performs the same amount of work as `ycsb-fg` but with many fewer tasks. Therefore, it consumes fewer on-chip commit queue resources, limiting the stalls and

aborts that occur when the commit queue is full. As a result, it performs about 2.9× better than Chronos-SLOT. This speedup is largely unaffected by the task-mapping strategy. The tasks are large and touch multiple objects, so specifying a single hint does not have significant impact.

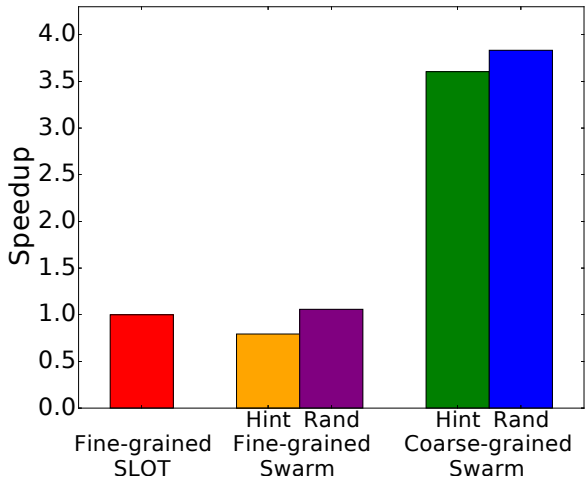


Figure 4-8: Performance of `ycsb` under Chronos-SLOT and different variants of Chronos-Swarm (specialized cores).

Figure 4-8 shows results with specialized cores. The pattern is largely the same. However, performance differences are larger; Chronos-Swarm with hint task mapping is 35% slower than Chronos-SLOT (vs. 7% with RISC-V cores). Specialized cores have higher throughput, so delayed conflict checks have a larger impact on the speculation window.

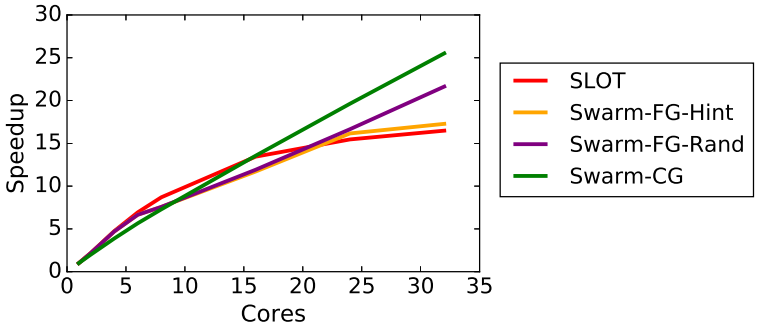


Figure 4-9: Self-relative scalability of `yscb` under different variants. All lines are normalized to its own single core runtime (RISC-V cores).

Scalability: Figure 4-9 shows the self-relative scalability of these variants. Each line is normalized to its own 1-core runtime. We only show one line for Swarm-CG because the task mapping strategy does not significantly affect performance with coarse-grained tasks.

SLOT and Swarm-FG-Hint show the worst scaling, only achieving a 16.5× and 17.3× speedup with 32 cores. These variants suffer from load imbalance issues. Swarm-FG-

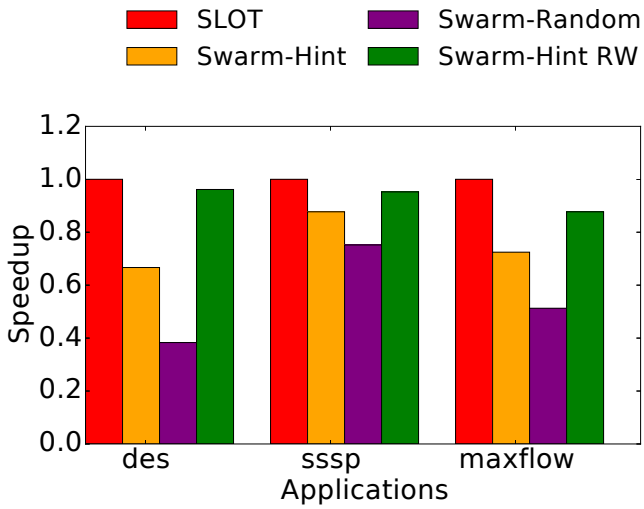


Figure 4-10: Performance of different variants of Chronos-Swarm in comparison to Chronos-SLOT under RISC-V cores.

Rand and Swarm-CG does not suffer from load imbalance, scaling to 21.6× and 25.5× respectively.

4.4.2 Performance of `des`, `maxflow`, and `sssp`

We now evaluate Chronos-Swarm with the applications used in Chapter 3 to evaluate Chronos-SLOT. All read-write data in these applications have high write intensity. Therefore, according to the discussion in Section 4.1.2, this data should be best served partitioned. These applications still access one read-write object per task, and therefore does not use Swarm's ability to access multiple objects per task. We perform these experiments to identify Swarm's overheads in comparison to SLOT.

First, we look at the performance of RISC-V implementations in Figure 4-10. For each application, Figure 4-10 shows the performance of the same accelerator under SLOT, Swarm with hint task mapping (Swarm-Hint), Swarm with random task mapping (Swarm-Random), and also another variant where Swarm-Hint is extended to perform conflicts checks only on read-write data (Swarm-Hint RW).

These results show two general trends. First, Swarm-Random is strictly worse than Swarm-Hint for all applications. This makes sense, since all tasks access a single object. Swarm-Random, unlike Swarm-Hint, can run tasks that access the same object on different tiles. This hurts locality and triggers additional coherence traffic.

Second, the difference between SLOT and Swarm-Hint is much larger than in `ycsb`. This is because, unlike `ycsb`, these applications have read-only data. All of these are graph applications, which use read-only arrays to represent the graph structure. Swarm performs conflict checks on accesses to these arrays, while SLOT does not. Ideally, these conflict checks should not cause aborts since they are all reads. However, with Chronos-Swarm, these accesses will get recorded in the Bloom filter, increasing the chance of false-positives. In addition, these conflict checks incur additional latency, keeping the

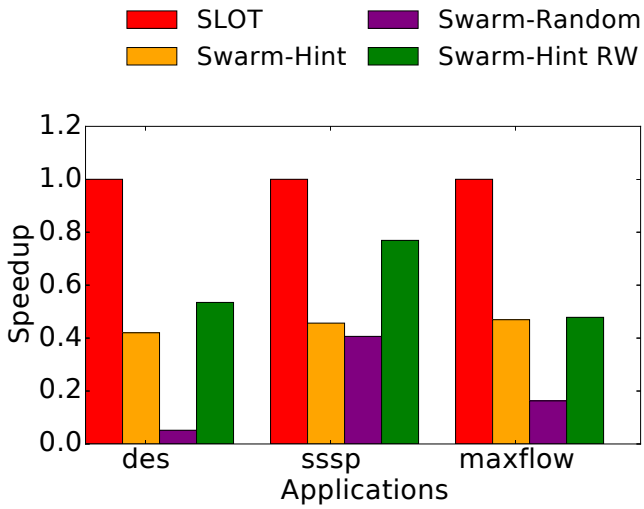


Figure 4-11: Performance of different variants of Chronos-Swarm in comparison to Chronos-SLOT under specialized cores.

core stalled.

To investigate the effect of these unnecessary conflict checks, we extended Chronos-Swarm to have configurable conflict detection regions. Specifically, we extend Chronos to support a run-time configurable parameter that allows the user to specify a memory range on which conflicts checks are performed. We see that this variant (Swarm-Hint-RW in Figure 4-10) gets back nearly all of the losses for `des` and `sssp`.

With `maxflow`, Swarm-Hint-RW is still 20% slower than SLOT. We find that this is because of the object granularity. `maxflow` uses large multi-word objects, with a single task performing many accesses to the object's data. With SLOT, each task performs only a single conflict check, which is done before the task starts. However, with Swarm, each memory access triggers its own conflict check, increasing latency.

Next, we look at the performance of specialized core accelerators for these applications in Figure 4-11. These follow the same pattern as in RISC-V cores. However, Swarm-Hint RW cannot recoup all the losses suffered when going from SLOT to Swarm-Hint, `maxflow` being the worst affected with a 2× slowdown. The reason for these slowdown are the same as with `ycsb`. Specialized cores have higher throughput and as a result suffers from Swarm's delayed conflict checks.

4.4.3 Bandwidth overheads

Figure 4-12 shows the memory bandwidth consumed for `sssp` specialized core variant with both SLOT and Swarm. We only show `sssp` because the other applications have a working set that fits in the cache and as a result consume little memory bandwidth.

SLOT consumes about 8000 MB/s of DDR bandwidth. With Swarm-Hint, this number increases by 51%, mainly due to the extra DDR traffic incurred with preserving sticky bits across directory evictions. While our FPGA platform supports a theoretical

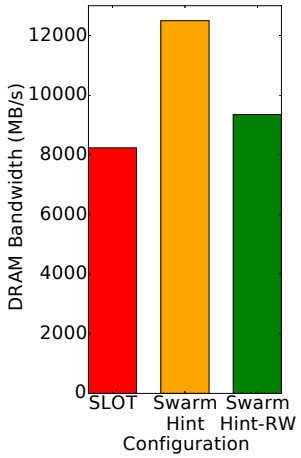


Figure 4-12: DRAM Bandwidth requirement for `sssp` with different configurations.

	LUTs	FFs	BRAM	URAM
Chronos-SLOT Tile	77.6K	50.0K	69	16
Chronos-Swarm Tile	84.0K	63.8K	85	16
Directory Bank	6.2K	5.7K	17	16

Table 4.2: Chronos-Swarm FPGA resource consumption in comparison to Chronos-SLOT

maximum of 50 GB/s bandwidth, we find that in practice this is much lower, reaching a maximum of only 12 GB/s. `sssp` reaches this limit, which is another contributing factor to its slowdown on Swarm.

When conflict checks are limited to only read-write data, sticky bits are only need to be preserved for a subset of cache lines. Therefore the bandwidth overhead drops below this limit. As a result, `sssp` has a higher benefit than the other applications from limiting conflict checks to read-write data.

4.4.4 Resource consumption

Table 4.2 shows a single tile FPGA resource consumption for Chronos-SLOT and Chronos-Swarm instances with RISC-V cores. Chronos-Swarm adds a new Address Set module to a tile, which increases the per-tile LUTs by 8%, flip-flops by 28%, and Block RAMs by 23%. This is a reasonable overhead compared to the generality and ease-of-programming that the Swarm execution model allows.

However, in addition to the overheads in a tile. supporting Swarm also requires coherence directories. The table also shows the resources required for a directory bank. A directory bank consumes about the same resources as a private cache; about 7% of the tile's LUTs, 20% of the BRAMs, and all of the URAMs. For systems with a large number of tiles, these overheads are small, but they could be expensive for systems with a small

number of tiles.

4.5 Conclusion

This chapter presented Chronos-Swarm, an extension to Chronos that support the Swarm execution model. The Swarm execution model allows tasks to arbitrary data. To support it, we extend Chronos with hardware to track the memory addresses that each task accessed, and on a new access, locate and abort all conflicting tasks.

We showed that Chronos-Swarm allows efficient implementations of new applications, with `ycsb` being 3.5× better on Chronos-Swarm compared to Chronos-SLOT. However, for applications that map well to SLOT, Swarm's overheads cause up to 2× hit in performance.

These overheads stem from dynamic costs, such conflict checks and additional memory bandwidth, but fixed costs are small, about 15% of the area overheads. This motivates a hybrid execution model that combines SLOT and Swarm, which we present next.

CHAPTER 5

A hybrid SLOT/Swarm execution model

5.1 Motivation

In Chapter 4, we extended Chronos to support the Swarm execution model. However, this came at a cost for tasks that do not require arbitrary memory accesses. Section 4.4 showed that Chronos-Swarm is slower and requires more main memory bandwidth for applications whose tasks have an efficient data-centric mapping to the SLOT execution model.

We note that the hardware structures used in Chronos-SLOT and Chronos-Swarm are largely similar and the two systems have comparable area. The differences between the two systems mainly arise from the different flow of data among these structures. A natural question arises as to whether we could have a hybrid execution model that supports both SLOT and Swarm semantics.

Such an execution model offers many advantages. Beyond reclaiming lost performance for fully data-centric applications, it also allows for incremental application development. An application developer could start with the easier-to-program Swarm model. They would use profiling tools to identify frequently accessed objects and refactor the program to isolate such accesses into SLOT tasks.

In this chapter, we propose such an execution model. We first describe its semantics, and then describe its Chronos implementation. We then illustrate how Chronos-Hybrid allows incremental application optimization with `ycsb`, and show that the hybrid model allows implementations that outperform both SLOT and Swarm.

		Can access?			
Task-type	object-id	A	B	C	D
SLOT	A	✔	✘	✘	✘
SLOT	B	✘	✔	✘	✘
Swarm	-	✘	✘	✔	✔

Table 5.1: Legal accesses for the hybrid execution model. Assume no SLOT tasks exists for objects C and D.

5.2 Hybrid SLOT/Swarm execution model

Similar to SLOT and Swarm, the hybrid execution model also consists of timestamped tasks. However, it allows a task to be enqueued with either SLOT or Swarm semantics. With a software interface, these tasks are communicated via the following methods:

```
slot::enqueue(task-type, timestamp, object-id, args...);
swarm::enqueue(task-type, timestamp, <hint>, args...);
```

SLOT and Swarm tasks may create children with either SLOT or Swarm semantics. However, there are restrictions on the data a task may access.

SLOT tasks have the same semantics as in Section 3.1.2. Each SLOT task must specify an object id when the task is created, and the task is only allowed to access read-write data that is deemed to be belonging to this object. Swarm tasks can access any data, but they are disallowed from accessing any data belonging to an object with a SLOT task. In addition, both types of tasks can access any read-only data. Swarm tasks may optionally be enqueued with a hint. This hint is only used for task mapping purposes and does not constrain the data the task can access.

Table 5.1 shows an example of these restrictions. Here, we consider four read-write objects A, B, C, and D. The programmer decides that A and B should follow a partitioned model, while C and D should be allowed to replicate. Therefore the program may create SLOT tasks with A and B as object id but not C or D.

SLOT tasks with A as its object id can only access A's data and a task with B as its object id can only access B's data. Any Swarm task can access data of object C or D, since they are not covered by any other SLOT task, but they cannot access data of either A or B.

5.2.1 Facilitating incremental application development with the hybrid execution model

The hybrid SLOT/Swarm execution model facilitates incremental application development and optimization. We illustrate this process through a concrete example on how `ycsb` can be optimized. Section 5.4 evaluates the performance impact of each of the following steps.

Step 1. Develop an implementation with all tasks using the Swarm execution

model: We start with a fully shared memory implementation where each `ycsb` transaction is run in a single Swarm task, similar to Figure 4-1(c). This is preferred as a starting point since Swarm does not impose any constraints on data a task may access.
Step 2. Identify data better served with a partitioned model: We perform runtime profiling on this implementation, and see that all accesses to a B-tree node below a certain level N have a low write intensity and high stack distance, and as a result, is better served under a partitioned model. Therefore, we break the all accesses to these nodes into fine-grained tasks. Note that all tasks are still Swarm tasks.
Step 3. Add hints to fine-grained tasks: We add hints to all fine-grained tasks. For `ycsb`, a good hint would be the node id.
Step 4. Turn fine-grained tasks into SLOT tasks: At this point, the fine-grained tasks satisfy all requirement of a SLOT task. We convert the task's hint to be its SLOT object id, and launch them as SLOT tasks.

5.3 Chronos-Hybrid implementation

We extend Chronos to support this hybrid execution model, which we call Chronos-Hybrid. Chronos-Hybrid extends Chronos-Swarm to support SLOT tasks. This does not add any additional hardware structures beyond what was described for Chronos-Swarm in Section 4.2. But it changes the flow of data between some structures to allow execution of SLOT tasks.

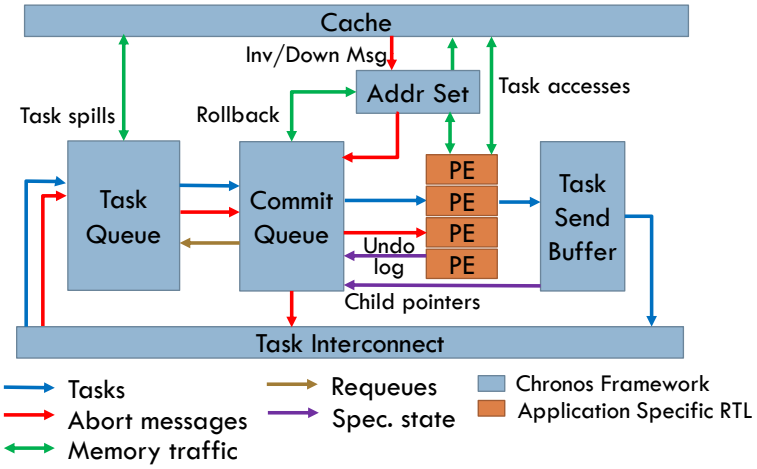


Figure 5-1: Tile microarchitecture for Chronos-Hybrid execution model. This is similar to the Chronos-Swarm microarchitecture. However, the commit queue now performs conflict checks for SLOT tasks before the task is dispatched, and all memory accesses from SLOT tasks bypass the Address Set module.

PE interface: Both SLOT and Swarm tasks use the same task signature. Therefore, Chronos-Hybrid only needs a single additional bit to specify whether a task is a SLOT task or a Swarm task. This bit is added to both the *Enq Child* and *Deq Task* ports described in Figure 3-8.

Intra-tile dataflow: The dataflow within a tile depends on the task type. The primary difference between SLOT and Swarm is whether a conflict detection takes place at dispatch time. SLOT tasks do this, Swarm tasks do not. Therefore, we modify the commit queue to check this condition before performing a conflict check. If the task is a SLOT task, the commit queue aborts all tasks with the same object id and with higher timestamp.

The commit queue also serializes SLOT tasks with the same object id and Swarm tasks with the same hint. While SLOT tasks must be serialized for correctness, for Swarm tasks, it is only an optimization. Chronos can be configured to disable serialization of Swarm tasks if desired.

While the task is running, a SLOT task can access any data without additional conflict checks. Therefore, such accesses bypass the Address Set module, and go directly to the cache. Accesses from all Swarm tasks are conflict-checked and go through the Address Set module.

Task mapping: Chronos-Hybrid maps SLOT tasks with the same object id to the same tile. For Swarm tasks, Chronos can be configured to either map them randomly or based on the hint.

Coherence protocol: The hybrid execution model does not allow the same data to be accessed from both SLOT and Swarm tasks. Therefore, we can guarantee that accesses for a given SLOT object will only come from the same tile. Such data need not participate in the coherence protocol. Accesses to such data bypasses the directory, and as storing sticky sharers is not required, they do not cause extra memory traffic.

However, a corner case can arise when a misspeculating Swarm task accesses read-write data that can also be accessed by a SLOT task. Under normal operation, when the Swarm task aborts, any writes it performed will be rolled back, and the conflict check on the rollback write will detect any tasks that may have read it. However, if a SLOT task happened to read this data before the rollback, then it will not be detected as a conflict, since SLOT tasks do not record their accesses.

Chronos-Hybrid prevents this situation by requiring that accesses from SLOT tasks be segregated in memory from accesses from Swarm tasks. Chronos-Hybrid requires that this SLOT region be communicated to the runtime in advance. All accesses from Swarm tasks are filtered to detect accesses to this region. If a Swarm task performs such an access, the task is immediately aborted.

5.4 Chronos-Hybrid evaluation

We implement Chronos-Hybrid on the same FPGA platform using the same parameters as in Table 4.1. We generate Chronos-Hybrid accelerators (with specialized cores) for the same applications and compare their performance with those generated from Chronos-SLOT and Chronos-Swarm.

5.4.1 Performance of Chronos-Hybrid

We first evaluate how `ycsb` performs under different Chronos variants and then move on to the other applications.

Performance of `ycsb`: Figure 5-2 shows how the performance of `ycsb` changes as we

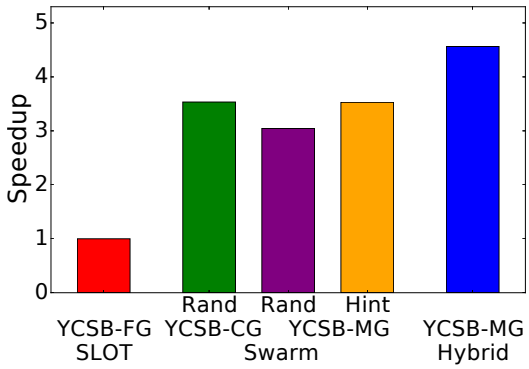


Figure 5-2: Performance of `ycsb` under the SLOT, Swarm and Hybrid execution models.

follow the incremental development approach highlighted in Section 5.2.1. All bars are normalized to the runtime of the SLOT implementation.

A coarse-grained `ycsb` Swarm implementation outperforms SLOT by 3.5×. This corresponds to step 1 of the incremental optimization process, and is the same result highlighted in Section 4.4. The performance difference arises because the coarse-grained implementation has lower tasks and suffers from lower stalls and aborts when the commit queue fills up.

Next, we create separate tasks for accesses to nodes in the two deepest levels. We call this implementation `ycsb-mg` (for mixed grain). This implementation creates additional tasks than `ycsb-cg`, so the performance decreases by 16%. However, when we add hints to the fine-grained tasks (step 3), this loss is recovered. In step 4, we turn the fine-grained into SLOT tasks. SLOT detects conflicts early, so it increases performance by 29% over the best Chronos-Swarm implementation.

Performance of the other applications: Figure 5-3 shows how the Chronos-Hybrid variant compares with Chronos-SLOT and Chronos-Swarm for `des`, `sssp`, and `maxflow`. For Chronos-Swarm, we compare with the best-performing variant where conflict checks are only performed on read-write data.

For these three applications, all tasks use a data-centric pattern, hence the performance is improved over Chronos-Swarm, with `des` and `maxflow` showing a 65% increase.

Ideally for `des`, `sssp`, and `maxflow`, Chronos-SLOT should have identical performance to Chronos-Hybrid. However, this is not the case in practice because of an implementation difference between the two variants. In order to reach timing closure, the Chronos-Hybrid specialized cores use more pipeline stages internally. This does not affect throughput, but slightly increases the latency of each memory access. This latency increase hurts performance by limiting the speculation window.

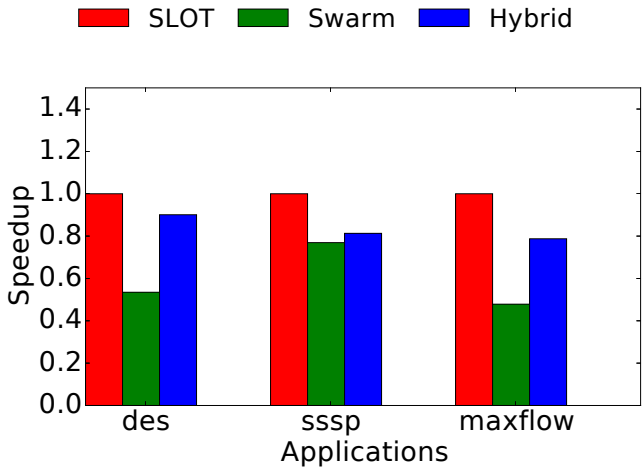


Figure 5-3: Summary of performance of `des`, `sssp`, and `maxflow` under different Chronos variants

Memory bandwidth: Figure 5-4 shows the main memory bandwidth consumption

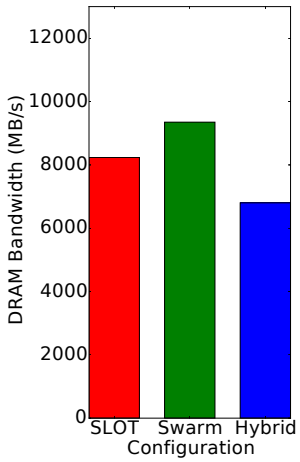


Figure 5-4: DRAM Bandwidth requirement for `sssp` under the SLOT, Swarm and Hybrid execution models.

of `sssp` under each of Chronos-SLOT, Chronos-Swarm, and Chronos-Hybrid. Despite achieving similar performance, the hybrid implementation requires 37% less bandwidth than Swarm. This is because Chronos-Hybrid does not require storing sticky sharers as in Chronos-Swarm and the bandwidth needed to preserve these bits across directory evictions is saved. Chronos-Hybrid requires less bandwidth than Chronos-SLOT because the runtime of Chronos-Hybrid is higher than that of Chronos-SLOT.

5.4.2 Area overhead of Chronos-Hybrid

Table 5.2 shows how the FPGA resource consumption for Chronos-Hybrid compares with the other two variants. These numbers are for a single tile with 8 RISC-V cores. However, the cores do not change across different variants so nearly all differences come from the framework components.

	LUTs	FFs	BRAM	URAM
Chronos-SLOT Tile	77.6K	50.0K	69	16
Chronos-Swarm Tile	84.0K	63.8K	85	16
Chronos-Hybrid Tile	88.3K	67.2K	94	16

Table 5.2: Per-tile Chronos-Hybrid FPGA resource consumption in comparison to Chronos-SLOT and Chronos-Swarm.

Chronos-Hybrid consumes about 5% more logic and about 10% more Block RAM than Chronos-Swarm. This increase is attributed to the commit queue, which now needs to store the object id of all SLOT tasks and perform dispatch-time conflict checks.

5.5 Conclusion

This chapter has presented a hybrid SLOT/Swarm execution model, that allows a task to follow either SLOT or Swarm semantics. SLOT tasks have more restrictions on the data they may access, but provide higher performance, while Swarm tasks have fewer restrictions on task accesses, but with more overheads.

We showed, using `ycsb`, that accelerators developed using this hybrid execution model can outperform accelerators developed using either SLOT or Swarm. This arises because different objects in the application may have different data access patterns, and the hybrid model allows developers to select the right task type based on the access pattern of the data they access.

CHAPTER 6

Conclusion

To conclude this thesis, we summarize our contributions and outline areas for future work.

6.1 Contributions

Current accelerators have been limited in the types of parallelism that they can exploit. They have mostly focused on domains where parallelism is easy to exploit, such as deep learning. Many applications do not have such easy-to-extract parallelism and have remained off-limits to accelerators.

This thesis has presented techniques to build accelerators for applications with speculative parallelism. These applications consist of atomic tasks, sometimes with order constraints, and need speculative execution to extract parallelism. For some applications, like discrete event simulation, speculation allows extracting parallelism using a work-efficient algorithm, but prior accelerators have had limited support for such task-level speculation.

Therefore, this thesis has made the following contributions:

- This thesis has presented Chronos, a framework-based approach for building accelerators that use speculation to extract parallelism. Under Chronos, accelerator designers express the algorithm as a set of ordered tasks, and then design processing elements (PEs) to execute each of these tasks. The framework provides reusable components for task management and speculative execution, saving most of the developer effort in creating accelerators for new applications.

- To address the challenge that prior speculation mechanisms are expensive for accelerators, this thesis has also proposed SLOT, a new execution model for speculative parallelism that does not require a coherence protocol. Instead, SLOT follows a data-centric approach, where shared data is mapped across the system; work is divided into small tasks that access at most one shared object each; and tasks are always sent to run at the place where their data is mapped.
- This thesis has presented an FPGA implementation of Chronos that uses commodity FPGAs in the cloud and demonstrates significant speedups for several challenging applications, including discrete event simulation and graph applications.
- This thesis has presented an extension to Chronos to support the Swarm execution model. We extend Chronos with a two-level coherence protocol, and implement address tracking structures to support tasks that access arbitrary data. We have shown that Chronos-Swarm allows of acceleration of new applications that would not be efficiently mapped to SLOT, but suffers from overheads for those applications that maps well to SLOT.
- Finally, this thesis has presented a hybrid SLOT/Swarm execution model that can capture the benefits of both SLOT and Swarm.

6.2 Future Work

The techniques outlined in this thesis open a number of avenues for future work:

- *Find and accelerate more work-efficient variants of existing workloads:* When multiple algorithms can be used to solve the same problem, current accelerators have often picked the one with easily extractable parallelism, even when it performs more overall work. The techniques outlined in this thesis could be used to extract parallelism using other more work-efficient algorithms.

  For example, we showed that in discrete event simulation, the techniques that yielded higher scalable implementations in software relied on the work-inefficient algorithm (Chandy-Misra-Bryant). In contrast, Chronos extracts parallelism from the more work-efficient algorithm, Time Warp.

  In circuit simulation, the current state-of-the art simulators such as Verilator [88], Cuttlesim [79], and ESSENT [8] evaluate each expression every cycle, regardless of whether the inputs changed or not. An event-driven approach would enable better work efficiency, but the overhead of tracking events can often outperform any gains in purely software simulators. Chronos provides support for event tracking, potentially making event-driven circuit simulation more favorable.

  Further, even most dynamical system simulators, such as Anton [86] in molecular dynamics, advances the state of all particles in a physical system with a

fixed timestep. An alternative approach is to use a per-particle adaptive timestep, where slower moving particles use a larger timestep. The latter is clearly more work efficient but is not easily parallelizable through conventional techniques, since only a few particles would have pending events at a given timestep. Speculative techniques can be used to extract parallelism across multiple timesteps. Therefore, Chronos enables exciting new research into such event-driven algorithms.

- *A silicon implementation of Chronos:* A silicon implementation of Chronos would allow for significant speedups over FPGA implementations. Ideally, such a silicon implementation should have a user-friendly, programmable interface for the algorithm designers to experiment more work-efficient algorithms that could be parallelized using speculative execution.

- *HLS tools to generate PEs:* The PEs for most Chronos applications have been generated using hand-written RTL. High-Level Synthesis (HLS) tools could be adapted to help with this process. However, current commercial HLS tools do not generate efficient PEs for irregular applications, which suffer from unpredictable memory latencies. A potential way around this would be to divide a task into small fixed-latency subtasks, which can be synthesizable using HLS, and then schedule them using Chronos's techniques.

- *Understanding the relationship between communication mechanisms and data dependences:* Spatial architectures, like systolic arrays or CGRAs, use a grid of PEs with nearest-neighbor communication using FIFO channels. In contrast, Chronos uses an all-to-all communication pattern with priority-ordered channels. Future work could explore more on the fundamental differences between these communication patterns, and how they are influenced by an application's data dependences.

Bibliography

[1] M. Abeydeera, M. Karunaratne, G. Karunaratne, K. De Silva, and A. Pasqual, "4K real-time HEVC decoder on an FPGA," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 1, pp. 236–249, 2015.

[2] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez, "SAM: Optimizing multithreaded cores for speculative parallelism," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[3] G. Al-Kadi and A. S. Terechko, "A hardware task scheduler for embedded video processing," in *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers*, 2009.

[4] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[5] R. J. Anderson and J. C. Setubal, "On the parallel implementation of Goldberg's maximum flow algorithm," in *Proceedings of the fourth annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[6] AWS FPGA Hardware and Software Development Kit, "https://github.com/aws/aws-fpga," 2017.

[7] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[8] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate RTL simulation," in *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[9] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[10] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proceedings of the IEEE Infocom*, 2000.

[11] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[12] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[13] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[14] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[15] I. Bratt, "The ARM® Mali-T880 mobile GPU," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015, pp. 1–27.

[16] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011.

[17] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A High-performance, Low Memory, Modular Time Warp System," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS)*, 2000.

[18] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[19] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[20] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 4, 1981.

[21] H. Chen, R. Lagerquist, A. Nayak, H. Mair, G. Manoharan, E. Wang, G. Gammie, E. Ho, A. Rajagopalan, L.-K. Yong *et al.*, "A 7nm 5G Mobile SoC Featuring a 3.0 GHz Tri-Gear Application Processor Subsystem," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2021.

[22] T. Chen, S. Srinath, and G. E. Batten, Christopher Suh, "An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware," in *Proceedings of the 51st International Symposium on Microarchitecture*, 2018.

[23] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[24] G. C. Chuang, P.-A. Ting, J.-Y. Hsu, J.-Y. Lai, S.-C. Lo, Y.-C. Hsiao, and T.-D. Chiueh, "A MIMO WiMAX SoC in 90nm CMOS for 300km/h mobility," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2011, pp. 134–136.

[25] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving DNNs in real time at datacenter scale with project Brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[26] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[27] V. Dasu, L. Sihao, and T. Nowatzki, "PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[28] C. Demetrescu, A. Goldberg, and D. Johnson, "9th DIMACS Implementation Challenge: Shortest Paths," 2006. [Online]. Available: http://www.dis.uniroma1.it/~challenge9

[29] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd Annual International Symposium on Computer Architecture (ISCA)*, 1974.

[30] T. Dias, N. Roma, and L. Sousa, "High performance multi-standard architecture for DCT computation in H. 264/AVC high profile and HEVC codecs," in *Proceedings of the IEEE Conference on Design and Architectures for Signal and Image Processing*, 2013.

[31] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol Verification as a Hardware Design Aid," in *Proceedings 1992 IEEE International Conference on Computer Design*, 1992.

[32] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[33] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: new frontiers in performance improvements and energy efficiency," *Intel white paper*, vol. 19, no. 20, 2008.

[34] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1198.

[35] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan, "Design and evaluation of the rollback chip: special purpose hardware for Time Warp," *IEEE Transactions on Computers*, vol. 41, no. 1, 1992.

[36] E. Gaona-Ramirez, R. Titos-Gil, J. Fernandez, and M. E. Acacio, "Characterizing energy consumption in hardware transactional memory systems," in *Proceedings of the 22nd Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 22)*, 2010.

[37] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[38] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[39] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *IEEE Pervasive Computing*, vol. 7, no. 4, 2008.

[40] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[41] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[42] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[43] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[44] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, 2010.

[45] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, 2014.

[46] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011.

[47] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[48] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[49] S.-A. Hwang and C.-W. Wu, "Unified VLSI systolic array design for LZ data compression," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 9, no. 4, pp. 489–499, 2001.

[50] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar, "Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[51] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time Warp Operating System," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987.

[52] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[53] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

[54] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[55] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez, "Harmonizing speculative and non-speculative execution in architectures for ordered parallelism," in *Proceedings of the 51st International Symposium on Microarchitecture*, 2018.

[56] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[58] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[59] H. Kung and C. Leiserson, "Algorithms for VLSI processor arrays," in *Introduction to VLSI systems*, 1980.

[60] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the Top: What will drive computer performance after Moore's law?" *Science*, vol. 368, no. 1079, June 2020.

[61] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[62] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[63] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[64] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges,

and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[65] K. Locke, "Parameterizable Content-Addressable Memory," *Xilinx Application Note*, 2011.

[66] S. Loncaric, "A survey of shape analysis techniques," *Pattern Recognition*, vol. 31, no. 8, pp. 983–1001, 1998.

[67] X. Ma, D. Zhang, and D. Chiou, "FPGA-Accelerated Transactional Execution of Graph Workloads," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[68] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[69] S. Margerm, A. Sharifian, A. G. Guha, and G. Shriraman, Arrvindh Shriraman Pokam, "TAPAS: Generating Parallel Accelerators from Parallel Programs," in *Proceedings of the 51st International Symposium on Microarchitecture*, 2018.

[70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[71] U. Meyer and P. Sanders, "Delta-Stepping: A Parallel Single Source Shortest Path Algorithm," in *Proc. of the 6th Annual European Symposium on Algorithms (ESA)*, 1998.

[72] G. E. Moore, "Cramming more components onto integrated circuits," in *IEEE Solid-State Circuits Society Newsletter*, 1965.

[73] K. Moore, J. Bobba, M. Moravan, M. D. Hill, and D. Wood, "LogTM: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[74] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st International Symposium on Microarchitecture*, 2018.

[75] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 40–50, 2017.

[76] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA framework for vertex-centric graph computation," in *Proceedings of the 22nd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014.

[77] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[78] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2011.

[79] C. Pit-Claudel, T. Bourgeat, S. Lau, and A. Chlipala, "Effective simulation and debugging for a high-level hardware language using software compilers," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 789–803.

[80] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: a reconfigurable accelerator for parallel patterns," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[81] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing," in *Proceedings of the 53rd International Symposium on Microarchitecture*, 2020.

[82] S. Rahman, N. Abu-Ghazaleh, and W. Najjar, "PDES-A: A Parallel Discrete Event Simulation Accelerator for FPGAs," in *Proc. of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, 2017.

[83] N. Ramanathan, G. A. Constantinides, and J. Wickerson, "Precise pointer analysis in high-level synthesis," in *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[84] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, "Thread-level speculation on a CMP can be energy efficient," in *Proceedings of the 19th Annual International Conference on Supercomputing*, 2005.

[85] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation," in *Proceedings of the 19th Annual International Conference on Supercomputing*, 2005.

[86] D. E. Shaw, J. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton *et al.*, "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[87] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[88] W. Snyder, "Verilator and SystemPerl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[89] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995.

[90] SpinalHDL, "A FPGA friendly 32 bit RISC-V CPU implementation," https://github.com/SpinalHDL/VexRiscv, 2018.

[91] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000.

[92] S. Subramanian, "Architectural techniques to unlock ordered and nested speculative parallelism," Ph.D. dissertation, Massachusetts Institute of Technology, 2018.

[93] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[94] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[95] E. Vasilakis, "An Instruction Level Energy Characterization of ARM Processors," *Technical Report FORTH-ICS/TR-450*, 2015.

[96] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[97] Y. Yang, W. Zhao, and Y. Inoue, "High-performance systolic arrays for band matrix multiplication," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2005, pp. 1130–1133.

[98] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[99] V. A. Ying, M. C. Jeffrey, and D. Sanchez, "T4: Compiling sequential code for effective speculative parallelization in hardware," in *Proceedings of the 47th International Symposium in Computer Architecture*, 2020.

[100] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, "Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

[101] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, 1998.

[102] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating large-scale single-source shortest path on FPGA," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 129–136.

[103] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, 2018, pp. 69–77.