Architectural Techniques to Unlock Ordered and Nested Speculative Parallelism

by

Suvinay Subramanian

Master of Science, Massachusetts Institute of Technology (2013)
Bachelor of Technology, Indian Institute of Technology Madras (2011)

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

Doctor of Philosophy
at the Massachusetts Institute of Technology

February 2019

Author . . . . . .
Department of Electrical Engineering and Computer Science
September 28, 2018

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

Architectural Techniques to Unlock Ordered and Nested Speculative Parallelism

by

Suvinay Subramanian

Submitted to the Department of Electrical Engineering and Computer Science
on September 28, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Current multicores suffer from two major limitations: they can only exploit a fraction of the parallelism available in applications and they are very hard to program. This is because they are limited to programs with coarse-grained tasks that synchronize infrequently. However, many applications have abundant parallelism when divided into small tasks (of a few tens to hundreds of instructions each). Current systems cannot exploit this fine-grained parallelism because synchronization and task management overheads overwhelm the benefits of parallelism.

This thesis presents novel techniques that tackle the scalability and programmability issues of current multicores. First, Swarm is a parallel architecture that makes fine-grained parallelism practical by leveraging order as a general synchronization primitive. Swarm programs consist of tasks with programmer-specified order constraints. Swarm hardware provides support for fine-grained task management, and executes tasks speculatively and out of order to scale. Second, Fractal extends Swarm to harness nested speculative parallelism, which is crucial to scale large, complex applications and to compose parallel speculative algorithms. Third, Amalgam makes more efficient use of speculation resources by splitting and merging address set signatures to create fixed-size units of speculative work. Amalgam can improve performance and reduce implementation costs.

Together, these techniques unlock abundant fine-grained parallelism in applications from a broad set of domains, including graph analytics, databases, machine learning, and discrete-event simulation. At 256 cores, our system is $40\times$–$512\times$ faster than a single core system and outperforms state-of-the-art software-only parallel algorithms by one to two orders of magnitude. Besides achieving near-linear scalability, the resulting programs are almost as simple as their sequential counterparts, as they do not use explicit synchronization.

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

Acknowledgments

Over the course of my PhD journey, I have been extremely fortunate in having had incredible mentors, colleagues, and friends who have helped me connect the dots along the way—scientifically, professionally and personally. I would like to express my sincere and heartfelt gratitude to all of them.

First and foremost, I owe thanks to my research advisor, Professor Daniel Sanchez. I switched advisors rather late in my graduate career. Daniel quickly brought me up to speed and guided me towards important and challenging problems. I have gained a deeper and more fundamental understanding of computer systems through my countless hours of discussions and interactions with him. Daniel taught me how to formulate research ideas, how to carefully, patiently, and rigorously develop them, and how to effectively communicate my work to the research community. Daniel's enthusiasm for diving into the details, his willingness to brainstorm or serve as a sounding board, anytime of the day or night, weekday or weekend, and his high standards have been critical to improving the quality and completeness of my research. But most importantly, Daniel developed in me the courage to work on difficult problems that few others in the world would venture to tackle. Daniel took me under his wing when my options were limited; this dissertation is my way of repaying that debt.

Apart from Daniel, I have had the pleasure of working with several other professors during my tenure at MIT. I have learned a great deal from their different styles, and also gained the confidence to pick up any new field and in great depth when required. I started my graduate career working with Professor Li-Shiuan Peh. Apart from her strong technical expertise, Li-Shiuan serves as a role model on how to effortlessly balance work and life, and remain ever ebullient and cheerful. I thank her for her mentorship, guidance and deep interest in my well-being many years after I switched advisors. After Li-Shiuan moved to Singapore, I worked briefly with Professor Hari Balakrishnan and Professor Mohammad Alizadeh. Thanks to Hari, for showing me how to pick problems at their infancy, and Mohammad for showing me how to extract the essence of a concept in the clearest form. Many thanks to Professor Anantha Chandrakasan, who not only welcomed me into his group, but provided valuable guidance and support when I was looking for a new advisor.

I am grateful to my thesis committee members Professor Joel Emer and Professor Srini Devadas. Joel has been deeply involved in all of the work presented in this dissertation, and brought an enormous degree of clarity to the concepts that we developed. Srini reminds me that systems research should be impactful in the real world, and serves as an inspiration by being a prolific researcher in a diverse array of fields.

This dissertation is the result of joint work with many collaborators: Mark Jeffrey, Cong Yan, Maleen Abeydeera, Hyun Ryong (Ryan) Lee and Victor Ying. Mark has been a partner-in-crime through several projects. I have learned a great deal from his attention to detail, software engineering practices, diligence, and relentless pursuit of perfection. Cong wrestled for several months with silo, one of the more notorious applications in our benchmark suite. Maleen patiently analyzed several applications, improved the accuracy of our simulator models, tuned our systems, and performed many experiments for our work. Ryan and Victor jumped onto the Fractal project as fresh graduate students and were instrumen-

tal in ensuring we completed the project in time for ISCA. I cannot wait to see how they further advance the ideas that we developed.

I am grateful to my several collaborators on prior work: Chia-Hsin Chen, Bhavya Daya, Tushar Krishna, Woo-Cheol Kwon, Sunghyun Park, Anirudh Sivaraman and Keith Winstein. It was a great pleasure sharing ideas and working together over the last several years. I am also thankful to the incredibly vibrant Sanchez group: Maleen Abeydeera, Nathan Beckmann, Nosayba El-Sayed, Yee Ling Gan, Harshad Kasture, Hyun Ryong Lee, Anurag Mukkara, Quan Nguyen, Po-An Tsai, Victor Ying, Cong Yan and Guowei Zhang. A lot of my learning in graduate school has come from the many hours of discussions, feedback and conversations with them. In addition to being the fiercest critics of my work, they have been incredibly fun to hang out with, go hiking, play board games and have engaging discussions on a plethora of topics. Thanks to the Julia group for accepting me as a pseudo-member in their lab space and bringing vitality to my work environment in my final year.

Maria Rebelo, Mary McDavitt and Sally Lee were phenomenal admin assistants, assisting with paper work, reimbursements, booking travel and conference rooms, and ordering cakes for my thesis defense. Janet Fischer and Professor Leslie Kolodziejski at the EECS graduate office patiently extended every PhD deadline, and provided helpful advice at various inflection points.

I would not have been able to withstand the vicissitudes of grad school without the army of friends who supported me, patiently endured and even encouraged my idiosyncrasies, and cheered my various endeavors both professional and personal. Anirudh has been a close friend and mentor through several years in grad school and undergrad. As he often remarks, our technical collaborations capture the quintessential collaborative spirit at MIT. More importantly, I am grateful for his friendship, and his valuable guidance and honest advice at critical junctures. Harshad and Tushar have been a sounding board, providing perspective, reassurance and confidence at many points. I am thankful for their words of wisdom and friendship. My long-term room mates Ujwal and Vaibhav made our apartment a home away from home. The MIT Ohms helped keep my passion for music alive, and provided many fun trips and cherished memories. Aditya, Abhinav, Anand, Anisha, Aritro, Arun, Ashwin, Atulya, Avinash, Chiraag, Deepak, Devendra, Divya, Giri, Jaichander, Karthik, Kaushik, Kumar, Krishna, Murali, Nachiket, Naveen, Nikhil, Parnika, Pranav, Radhika, Raghav, Rahul, Rik, Rohan, Rutu, Sai, Sameer, Sarvesh, Sivaraman, Srini, Sriram, Swati, Tapovan, Tejas, Varun, Vipul, Yashovardhan, and several more friends in Boston, the US and India: thank you for being the kindest, supportive and most wonderful friends. Krithika has been a great source of good cheer in the past year. A special thanks for her patience, support, and belief in me. Her unbounded enthusiasm and gaiety infused in me a sense of joy, optimism and passion for life.

My brother, Shashank, has consciously or unconsciously helped me keep my sanity by engaging my varied interests and hobbies from across the globe day after day, and with his ever-jovial attitude. I am grateful for his love and companionship. Finally, and most importantly, I am deeply, deeply grateful to my parents. Without their sacrifices over the years, the value they placed on education, the ethics and values they taught me, and their unwavering support, encouragement and love, I wouldn't be where I am today. This dissertation is dedicated to them.

Contents

List of Figures

List of Tables

■ 16

1

Introduction

PARALLEL machines are pervasive today—ranging from tiny computers that power smartphones, to high-end computers in data centers and supercomputing clusters. These machines can have hundreds of cores and provide copious compute capability, up to several teraflops of performance [149] on a single chip.

Yet, only a narrow sliver of individual applications can exploit this available parallelism. This is because current architectures are limited in the kinds of parallelism that can be effectively *expressed* and *exploited*. Modern computer systems exploit parallelism from problems at two extremes of granularity. At one extreme, systems exploit parallelism at the granularity of individual instructions through techniques such as out-of-order and superscalar execution. This is commonly referred to as instruction-level parallelism (ILP), and has been the predominant driver of performance improvements over the last three decades. ILP, however, is restricted by control flow and data dependences. Further, the hardware costs needed to exploit ILP—such as the instruction scheduling logic and wide register files—grow quadratically, and hence are not scalable [116].

At the other extreme, systems extract parallelism from applications that can be cast as coarse-grained tasks (several thousand to a few million instructions) with few data dependences and relatively infrequent synchronization. But such coarse-grained parallelism is also narrow in scope. It fares well in domains such as dense linear algebra and graphics, where applications have well-defined accesses to data and few unpredictable control and data dependences. By contrast, applications in several domains such as graph processing, sparse linear algebra, and discrete-event simulation are ill-served by coarse-grained parallelism. These applications often rely on pointer-based data structures, and data dependences are dictated by run-time data values that are difficult to predict statically and a poor match for current techniques.

Between these two extremes lies a wide spectrum of parallelism that has received little attention. Many applications have substantial parallelism when expressed as fine-grained tasks of a few tens to hundreds of instructions. Unfortunately, fine-grained tasks cause large overheads in current systems, because software task management overheads (e.g., schedul-

ing and load balancing) overwhelm the benefits of parallelism. At the same time, exposing such fine-grained parallelism often necessitates frequent and complex synchronization—for instance, to handle producer-consumer relationships, or to coordinate accesses to individual elements of a data structure. Current systems, however, lack efficient support for such synchronization.

As a result, today's programmers are in an unenviable position. While, sequential programming is simple and natural, ILP has been yielding diminishing returns over the last decade [16, 103]. On the other hand, parallel programming often requires heroic programming efforts to overcome the deficiencies of current architectures. And yet, these efforts may not yield concomitant performance gains. Even worse, the limits of current architectures constrain the imagination of algorithm designers and programmers. It biases them to only consider algorithms with parallelism that can be exploited by current architectures, while ignoring more efficient algorithms.

To resolve this conundrum, future parallel systems must provide greater *architectural support for parallelization*. This dissertation investigates what essential architectural support is required in order to mine the abundant, yet untapped, fine-grained parallelism in applications. Fundamentally, parallelizing a program consists of two main steps: dividing work into tasks that may run concurrently, and specifying the synchronization of tasks with potential data dependences to ensure correct behavior. The architecture of a parallel machine impacts and constrains both the manner and efficacy of these steps. It determines what forms of parallelism can be *expressed*, how effectively the parallelism is *exposed*, and the ability of the machine to exploit this parallelism and *scale* to a large number of processors. This not only influences how programs are written (or how code is compiled), but also affects the programmer's choice of data structures and algorithms.

This dissertation advocates that *parallel architectures should provide support for order as a general synchronization primitive among fine-grained tasks*. This is for two reasons. First, many applications require tasks to execute in a certain order to satisfy the semantics of the algorithm. Enforcing this order in software for fine-grained tasks incurs large overheads to be practical. Second, support for order naturally captures several complex synchronization patterns. This support enables parallelizing a wide range of general-purpose applications. With this architectural support, we show that it is possible to extract parallelism even in the presence of ordering constraints.

1.1 Challenges

We categorize the challenges to successfully exploiting parallelism along two dimensions: (1) expressing and extracting parallelism, and (2) reducing overheads and improving efficiency. Current parallel systems are beset with challenges along both these dimensions, severely impairing their ability to parallelize a wide spectrum of applications.

1.1.1 Expressing and Extracting Parallelism

A parallel architecture should ideally allow programmers to easily and effectively express parallelism of all forms and at all levels. This requires architectural support that is sufficiently general to capture many different forms of parallelism in applications. In addition, the architectural mechanisms must scale to a large number of cores.

Providing general support for synchronization: Parallelism arises in many flavors, such as data parallelism, task parallelism, pipeline parallelism and more. A common challenge in exploiting the various forms of parallelism is providing suitable abstractions for expressing synchronization among tasks. There exist several mechanisms in literature, such as locks, barriers, atomics, and producer-consumer relationships, to express such synchronization. Among these, a very general form of synchronization is *order*. With ordered execution, the tasks in a parallel architecture appear to run in an order (total or partial) specified by the programmer, even though the architecture may run tasks in parallel. Ordered execution is less restrictive than the sequential execution model. Ordered execution allows tasks to be created out of order and in parallel, and relies on the architecture to enforce order. By contrast, the sequential execution model consists of a single, serial stream of instructions or tasks. These form long chains of dependences that preclude them from being created or executed widely out of order.

Order confers several benefits. First, it is expressive enough to capture several complex synchronization patterns. In addition, many applications require tasks to execute in a certain order to satisfy the semantics of the algorithm (Section 3.1). Second, using order allows programmers to write and reason about programs almost like a serial program. This implicit approach means that the programmer need not, in general, determine whether parallelism exists among tasks nor consider how to explicitly express it. While it may seem that order limits parallelism, there are often opportunities for parallelism among ordered tasks that can be exploited with suitable architectural techniques. Unfortunately, current architectures provide limited support for expressing order and extracting parallelism under order constraints.

Extracting parallelism in the presence of ordering constraints often necessitates support for speculative execution. Speculative execution allows a processor to execute tasks out of order in the presence of control and data dependences that are unknown statically. It enables the processor to seize opportunities for parallel execution without violating true dependences among tasks. We find that ordering constraints are often specified conservatively, and can be elided subject to true dependences between tasks (Section 3.1). In many applications, these dependences manifest rarely, but can only be resolved at run-time, necessitating speculation to uncover parallelism.

Attempts at performing fine-grained ordered speculation in software have yielded scant benefits due to large overheads [100]. Hardware support for speculation has been provided in thread-level speculation (TLS) and transactional memory (TM) systems. However, the execution model for both TM and TLS are limited. TM does not provide support for order. The TLS execution model targets sequential programs which unnecessarily constrains par-

allelism in many applications; exposing order in the execution model, by contrast, allows creating tasks out of order exposing greater parallelism. In addition, both TM and TLS suffer from implementation challenges: TM suffers from poor performance on fine-grained tasks, while TLS mechanisms scale poorly beyond a few cores (Section 2.4).

Providing modular composability: Parallelizing complex real-world applications is extremely challenging. Complex programs consist of several layers of interacting components, and any given procedure may invoke several nested procedures at other components. These procedures are often parallelizable. To harness all this parallelism, a parallel system should support *nested parallelism*, i.e., the ability to invoke a parallel algorithm within another parallel algorithm. Nested parallelism confers two major benefits. First, it lets the system exploit parallelism at all levels of the application, dramatically improving scalability. Second, it makes parallel software easy to compose, making it easy to write large, modular parallel programs.

Nested parallelism is a staple of many parallel languages that extract non-speculative parallelism, such as NESL, Cilk, and OpenMP [27, 81]. However, nested parallelism has been difficult to support in systems that perform speculative parallelization. Nesting in transactional memory systems [20, 206] focus on parallelizing at the coarsest (shallowest) levels, incurring large overheads that squander their potential. Further, to guarantee atomicity, nested transactions merge their speculative state with their parent, which is often at odds with the gains realized by parallelization. Finally, transactional memory systems are unable to express all forms of nested parallelism in applications, such as ordered parallelism.

1.1.2 Reducing Overheads and Improving Efficiency

Scaling to large systems introduces unavoidable latencies and imposes bandwidth constraints. Parallel systems must provide low-overhead mechanisms for communication and synchronization. Further, resource management strategies are also crucial to achieve good utilization and scalability.

Supporting fine-grained tasks: Many applications have substantial parallelism when expressed as fine-grained tasks of a few tens to hundreds of instructions, and are also more naturally expressed in that form. However, the overheads of managing and scheduling such fine-grained tasks in software can be large. Queue operations and synchronization through a shared-memory cache hierarchy incur significant latency, thwarting the benefits of fine-grained parallelism and limiting speedup. Prior work [120, 181] has shown that performing task management in hardware can yield significant benefits. But this has largely been explored in the context of non-speculative systems. Hardware task management in the speculative context acquires new dimensions—be it for correctness, performance or scalability. It necessitates new hardware structures (for instance, to manage tasks that may be aborted due to dependences), different policies (for scheduling or load balancing) and unique considerations (such as ensuring forward progress, prevention of deadlock and live-

lock). Unfortunately, prior work on hardware task management fails to address these concerns adequately.

Exploiting locality: To scale effectively, parallelism must not come at the expense of locality: tasks should be run close to the data they access to avoid global communication and use caches effectively. The need for locality-aware parallelism is well understood in non-speculative systems, where prior work has developed programming models to convey locality, and runtimes and schedulers to exploit it. By contrast, most prior work in speculative architectures has ignored the need for locality-aware parallelism: in TLS, speculative tasks are executed by available cores without regard for locality [94, 170, 191]; in TM, prior work has developed transactional task schedulers that limit concurrency to reduce aborts under high contention [14, 23, 221]. Limiting concurrency, i.e., controlling when to run tasks, suffices for small systems. However, scaling to hundreds of cores also requires solving the spatial mapping problem, i.e., controlling where to run speculative tasks, scheduling them across the system to minimize data movement.

Along with spatial mapping, it is also important to address load balancing. Load balancing improves the distribution of tasks across multiple cores, ensuring good utilization, and consequently better scalability. Load balancing has been studied extensively in the context of non-speculative systems [2, 2, 29, 91, 179, 220]. However, non-speculative approaches, such as work-stealing, work poorly in the speculative context because the signals to detect imbalance are different (Section 3.10).

1.2 Contributions

This dissertation presents novel architectural techniques to unlock ordered and nested speculative parallelism. Specifically we make the following contributions:

- *Swarm* is a scalable architecture for unlocking *ordered* parallelism—a type of parallelism that is abundant in many domains, such as simulation, graph analytics, and databases, but has been difficult to mine with current software and hardware techniques. Swarm successfully scales several challenging applications up to 256 cores while retaining the programming simplicity of their sequential counterparts.
- *Fractal* is a new execution model that enables seamless composition of ordered and unordered speculative parallelism. Our key insight is to employ order to guarantee atomicity of nested tasks rather than building impractically large atomic units.
- *Amalgam* is a microarchitectural technique that enables matching the speculation resources allotted to a task, to the granularity of the task. It ensures that speculation handles diverse task types across applications gracefully without impacting performance, while providing area savings.

These techniques were developed collaboratively as part of a multi-year effort involving several graduate students. The individual chapters detail the specific contributions of this dissertation in context.

Swarm: Programs with ordered parallelism have three key features. First, they consist of tasks that must follow a total or partial order. Second, tasks may have data dependences that are not known a priori. Third, tasks are not known in advance. Instead, tasks dynamically create children tasks and schedule them to run at a future time.

We design Swarm (Chapter 3), an architecture that successfully mines ordered parallelism. Swarm relies on a co-designed execution model and microarchitecture to scale efficiently. Swarm executes tasks speculatively and out of order, and efficiently speculates thousands of tasks ahead of the earliest active task to uncover enough parallelism. Swarm adapts existing eager version management (i.e., store speculative data in place and log old values) and conflict detection schemes [219], and contributes several new techniques that allow it to scale to large core counts and speculation windows.

In particular, Swarm's execution model conveys order constraints to hardware without undue false data dependences. Swarm's conflict detection scheme leverages eager versioning to, upon mispeculation, selectively abort the mispeculated task and its dependents, enabling greater scalability. We also design a distributed commit protocol that allows ordered commits without serialization, supporting multiple commits per cycle with modest communication. Finally, we develop *spatial hints*, a technique that conveys program knowledge to exploit locality and enable high-quality task mappings. We enhance spatial hints through a novel data-centric load balancer for the speculative context that redistributes tasks in a locality-aware fashion.

Together, these techniques enable Swarm to scale 10 challenging applications near linearly up to 256 cores, outperforming state-of-the-art software parallel implementations by $3 - 18\times$. Besides achieving near-linear scalability on algorithms that are often considered sequential, the resulting Swarm programs are almost as simple as their sequential counterparts.

Fractal: Large, complex programs have speculative parallelism at multiple levels and often intermix ordered and unordered algorithms. Speculative architectures should support composition of ordered and unordered algorithms to convey all this nested parallelism without undue serialization.

Fractal (Chapter 4) presents a new execution model for nested speculative parallelism. Fractal programs consist of tasks located in a hierarchy of nested *domains*. Within each domain, tasks can be ordered or unordered. Any task can create a new subdomain and enqueue new tasks in that subdomain. All tasks in a domain appear to execute atomically with respect to tasks outside the domain. Fractal allows seamless composition of ordered and unordered nested parallelism.

Our Fractal implementation builds on the Swarm architecture and supports arbitrary nesting levels cheaply. Unlike prior work, our implementation focuses on extracting parallelism at the finest (deepest) levels first. It guarantees the atomicity of large domains by enforcing an order among individual tasks rather than building impractically large atomic units. This allows the system to manage tiny speculative tasks that are easy to track in hardware, reaping the benefits of fine-grained parallelism, and avoids the problems that plague

prior nested-parallel HTMs. Together, our contributions make nested parallelism practical in speculative systems.

Fractal uncovers abundant fine-grained parallelism and scales challenging programs that have eluded parallelization for decades. In particular, Fractal is the first architecture that scales the full STAMP suite [134] to hundreds of cores, achieving gmean speedup of 177× at 256 cores and outperforming prior architectures by up to 88×.

Amalgam: Real-world applications are complex and diverse, featuring tasks of varying sizes. Systems that support speculative parallelization employ probabilistic structures such as *Bloom filters* to track the address sets of tasks using a bounded amount of hardware state [41, 42, 219]. These structures are conservatively sized at design time to handle commonly occurring task sizes. This can, however, lead to inefficient utilization for tiny tasks, while also offering no protection against high false conflict rates for applications with very large tasks.

Amalgam builds on the Swarm architecture and develops microarchitectural mechanisms to match Bloom filter resources to the size of the task. On the one hand, large address sets are tracked cooperatively by multiple Bloom filters. On the other hand, multiple tiny address sets are tracked in concert by a single Bloom filter when possible. Amalgam offers a new architectural choice unavailable in prior speculative systems: trading off efficiency of tracking address sets against the degree of ordered speculation at run-time.

Unlike prior speculative architectures, Amalgam gracefully responds to varying task sizes, improving performance and efficiency. At 256 cores, Amalgam can reduce the area consumed by Bloom filters by 4× while slightly improving performance.

MORE than a decade after the transition to multicore processors, we remain saddled with the legacy of sequential architectures developed for the uniprocessor context. It is imperative that we break free of this mold if we are to realize the promise of parallel computing. We must systematically explore and rigorously enable parallel systems to both *express* and efficiently *exploit* all forms of parallelism available in applications. The techniques presented in this dissertation represent an important step in this direction. They are a culmination of a vertically-integrated approach of studying applications and hardware architectures in concert. These techniques help bridge the *semantic* gap between hardware and software in current parallel systems, and enable us to realize a new breed of parallel architectures that are more versatile, easier to program, and scale to larger systems sizes efficiently.

2

Background

THIS chapter presents an overview on parallelism in applications and techniques to extract these forms of parallelism. We focus on literature that is generally related to the broad themes of this dissertation. Later chapters discuss literature that is specific to the techniques presented in those chapters.

2.1 Overview

Fundamentally, parallelizing an application consists of two main steps: organizing the computation of the application into a set of tasks that may execute concurrently, and specifying the synchronization of tasks with potential data dependences to ensure correct behavior. We use the term task to refer to a unit of computational work (for example, a short function, or a set of instructions). The architecture of a parallel machine impacts and constrains both the manner and efficacy of these steps. The key to building a versatile parallel architecture is to design an execution model that allows programmers to effectively express parallelism of all forms and support it with an efficient implementation.

In this chapter, we first describe what kinds of parallelism exist in applications (Section 2.2). While parallelism may be classified in many ways, we present a broad taxonomy that illustrates the challenges in expressing and extracting different forms of parallelism and informs the architectural support needed to effectively exploit them. We then describe the range of techniques to extract those forms of parallelism (Section 2.3). And finally, we review prior architectural support for parallelization and their associated successes and limitations (Section 2.4).

2.2 Parallelism in Applications

We categorize parallelism in applications along two dimensions—*regularity* and *granularity*. The first dimension captures how much information we have about the tasks in

the application statically, which informs the degree of run-time support needed to effectively parallelize the application. The second dimension captures the size of the tasks in the application, which informs how much hardware support we need to extract parallelism effectively.

2.2.1 Regularity of Parallelism

Regular parallelism: Applications have regular parallelism when their tasks and data dependences among them are known statically. Regular 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. Regular parallelism also manifests in applications with pipeline parallelism. Here computation is partitioned into a sequence of stages with each stage producing data consumed by the following stage. Stages may execute in parallel communicating shared data in a deterministic producer-consumer style. Regular parallelism also arises in other domains such as stencil computation [174] and fast Fourier transform (FFT) [166].

Irregular parallelism: Applications have irregular parallelism when their tasks or data dependences among them are unknown statically.[1] Instead, tasks create new tasks at run-time as they find more work to do. Further, the tasks may have data dependences that again manifest at run-time. Irregular parallelism arises in applications that have unpredictable control flow, or operate on data structures accessed in complex non-predictable patterns such as trees and graphs. Examples include Dijkstra's algorithm [58, 80], Kruskal's minimum spanning forest algorithm [58], and discrete-event simulation [172].

2.2.2 Granularity of Parallelism

Granularity refers to the size of tasks. Parallelism in applications can arise at various granularities: ranging from single-instruction tasks (instruction-level parallelism), to fine-grained tasks of a few tens to hundreds of instructions each, to coarse-grained tasks of hundreds of thousands of instructions each. While finer granularities can expose greater parallelism, they also impose larger costs—specifically, task creation, communication, and synchronization costs. Thus, the granularity of tasks impacts how much hardware support is required to get good performance.

[1] Irregular parallelism has a different connotation in the GPU literature, referring to unstructured computation with varying amount of work or unstructured memory accesses.

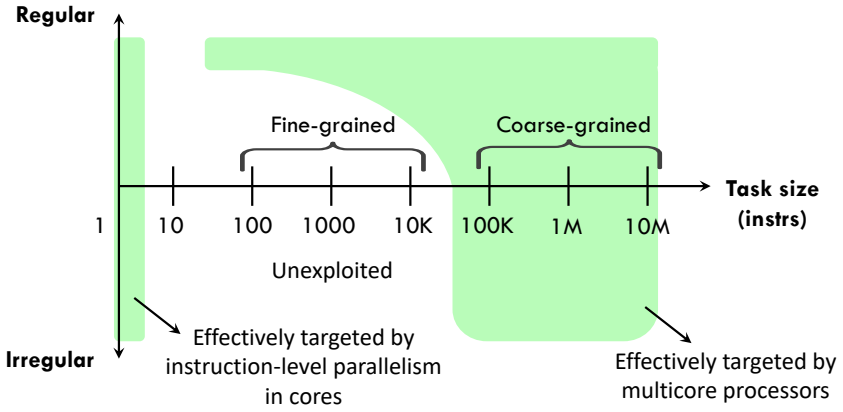


Figure 2-1: Granularity (X-axis) and regularity (Y-axis) are closely intertwined in determining the difficulty of extracting parallelism. Instruction-level parallelism of all forms is effectively exploited modern cores. Multicores can extract coarse-grained parallelism well. However, irregular fine-grained parallelism is taxing to extract on current systems.

Regularity and granularity are closely intertwined in determining the difficulty in extracting parallelism. With regular parallelism, we have advance information of what tasks need to run and when. This enables greater compile-time optimization—such as generation of parallel schedules before running the application—and therefore incurs fewer run-time costs. Irregular parallelism, by contrast, demands greater run-time support to discover and exploit parallelism. Likewise, the cost of managing coarse-grained tasks in software can be amortized and thus imposes lower overheads to parallelization. By contrast, managing fine-grained tasks demands greater hardware support.

Figure 2-1 illustrates the difficulty in extracting different forms of parallelism (Section 2.3 describes specific techniques). Current multicores exploit coarse-grained regular parallelism very well, enabled by programming languages, compiler tools and architectural support. Extracting coarse-grained irregular parallelism has seen limited success, the primary challenges being task management and synchronization. At the other extreme, modern cores effectively target instruction-level parallelism (ILP) in applications; but ILP is ill-suited for multicores. However, fine-grained irregular parallelism has been taxing to extract on current systems. Software-only approaches incur large overheads to be practical. At the same time, current architectural support is insufficient to mine such parallelism.

As we scale to larger multicores, it becomes imperative to extract all available forms of parallelism in applications. Amdahl's Law [9] states that maximum speedup achievable is limited by the non-parallelizable portion of an application. Given N cores, if the parallelizable fraction of the application, f, is accelerated by a factor N, then the speedup, S, is given as: $S = \frac{1}{(1-f)+f/N}$. As N increases, the speedup is quickly bounded by $\frac{1}{1-f}$. At $N = 100$ cores, even if 99% of the application is parallelizable ($f = 0.99$), $S = 50$, and if the fraction drops to even 95% ($f = 0.95$), the speedup drops precipitously to $S = 17$. It is therefore important to focus on even the hardest forms of parallelism, if we are to realize the benefits of parallel computing. Hence, this dissertation targets fine-grained irregular parallelism.

2.3 Techniques to Express and Extract Parallelism

2.3.1 Parallelizing Applications With Regular Parallelism

For applications with regular parallelism, it is possible to delineate tasks to be executed in parallel in advance. This may be done manually by the programmer or inferred automatically by the compiler.

Programming language extensions allow programmers to express data parallelism explicitly using constructs such as `parallel-for` in OpenMP [150]. Special purpose languages, such as ZPL [44], have also been proposed for writing data parallel programs. Language extensions such as Cilk [81] enable programmers to express task parallelism in applications.

Compiler tools rely on building and analyzing the program dependence graph [75] of the application. The program dependence graph captures control and data dependences among tasks of an application. Techniques such as *points-to analysis* [12, 51, 60, 73, 193] and *shape analysis* [47, 63, 87, 92, 123] inspect the program dependence graph and identify tasks that are independent and hence can be executed in parallel. Using these techniques, compilers can analyze and transform sequential programs into parallel ones.

The runtime systems for these applications tend to be simple, focusing primarily on load-balancing and priority scheduling of tasks on the critical path [29, 34].

Architectural support for single-instruction multiple-thread (SIMT) execution [129], common in GPUs, enables better scheduling of coarse-grained parallel tasks. While support for vectorization [175] and single-instruction multiple-data (SIMD) extensions (such as SSE and AVX [109]), enable extracting regular data parallelism at the granularity of individual instructions.

2.3.2 Parallelizing Applications with Irregular Parallelism

Parallelizing applications with irregular parallelism requires greater run-time support. The program dependence graph cannot be determined at compile-time, so static analysis fails to find much parallelism in these applications. Run-time approaches to parallelization may be categorized into two types: *speculative* and *non-speculative*.

Non-speculative approaches: Many non-speculative approaches rely on the programmer to explicitly delineate tasks and handle synchronization through constructs such as locks, semaphores, etc. Unfortunately, such manual code parallelization is a time-consuming and error-prone process. Programmers need to carefully tradeoff parallelism against ease of programming and performance [105]. Further, explicit synchronization techniques such as locks are not composable and require programmers to strictly adhere to locking conventions to prevent deadlock and livelock [96]. Finally, in many programs such explicit synchronization can be conservative because many dependences manifest infrequently at run-time [165, 207].

Software libraries and frameworks can abstract the implementation details from the programmer. Concurrent data structure libraries [37, 102, 105, 128, 198] provide tuned concurrent implementations of commonly used data structures such as stacks, lists and queues. Frameworks such as Galois [142, 155], Ligra [185], and GraphIt [223] focus on providing suitable abstractions for expressing irregular parallelism in applications, while handling synchronization and load-balancing under the hood.

Techniques such as *inspector-executor* [156] and deterministic reservations [25] take a different approach, building a dependence graph at run-time and executing tasks in parallel suitably. The inspector-executor approach builds the dependence graph and determines a schedule for tasks in an inspector phase, followed by an executor phase that executes the tasks in parallel. Deterministic reservations *reserve* access to data by different tasks in a priority order, followed by a *commit* phase where tasks with successful reservations execute in parallel and then free reservations to reserved data. These approaches work well with coarse-grained tasks where the run-time overhead of discovering parallelism can be amortized by the parallel execution of tasks.

Speculative approaches: 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 results of speculative execution are *committed*. In case of a violation, the offending tasks are *aborted* and re-executed. All speculative systems implement the following basic mechanisms: (1) *conflict detection* to track dependence violations, (2) *version management* to manage the speculative memory state of a task, and (3) *conflict resolution* to determine which task(s) to abort on a dependence violation.

Speculative parallelization is employed widely to exploit parallelism at the granularity of individual instructions in modern processors. Programmers simply write sequential code. Compilers can analyze the program to expose parallelism, generate an execution schedule and optimize branches [182]. Architectural support for ILP (Section 2.4.1) can issue and execute multiple instructions in parallel by performing dynamic dataflow analysis among instructions, handling operations with variable latency, predicting outcomes of branches, etc. [190].

Speculative parallelization has also been employed to parallelize applications with coarse-grained tasks, in domains ranging from transactional databases [121, 154, 205] to graph analytics [142] and discrete-event simulation [77, 112]. However, for applications with fine-grained tasks of a few tens to hundreds of instructions, the overheads of speculative parallelization in software are too large to be practical [101]. Further, many software techniques (e.g., ordered commits [100]) induce unnecessary serialization, limiting scalability even with plentiful speculative parallelism.

Architectural support for speculative parallelization can tackle some of these overheads. Prior work has proposed hardware techniques for supporting thread-level speculation (TLS) of sequential loops [94, 170, 191], and hardware transactional memory to perform speculative synchronization [42, 95, 104, 162]. However, both the TLS and TM execution models

are insufficient for many applications with irregular parallelism (Section 2.4). Further, TLS mechanisms often scale poorly beyond a few cores, while TM performs poorly with fine-grained tasks.

2.4 Architectural Support for Parallelization

2.4.1 Instruction-Level Parallelism

Support for instruction-level parallelism (ILP) enables systems to issue multiple independent instructions in the same cycle. Modern processors aggressively exploit dynamic ILP through a slew of techniques such as instruction pipelining [38, 201], superscalar execution [8, 189], register renaming [89], and out-of-order execution [203].

However, ILP has been yielding diminishing returns in recent years—from delivering 52% annual growth in performance during the 1990s, to a modest 3.5% annual growth in performance in the late 2000s [72, 103]. Control flow in many programs is complex and data-dependent. Memory aliasing and branch prediction are challenging to resolve despite sophisticated techniques. These issues fundamentally limit the number of instructions that may be issued in parallel [122, 158, 211]. Current systems only execute one to four instructions in parallel per cycle on average. Further, the instruction scheduling logic, register file, and other hardware structures to exploit ILP grow super-linearly as the number of execution units increases and with larger instruction windows [116]. These structures increase the area and power consumption, and are harder to engineer at high clock speeds, limiting overall performance. Consequently, ILP remains most suited to extracting parallelism across the execution units of a single core.

By contrast, Swarm (Chapter 3) operates at the granularity of fine-grained tasks of a few hundred instructions. Swarm hardware efficiently manages thousands of fine-grained tasks distributed across several tiles of a multicore. We provide high-throughput ordered commits of tasks in a distributed fashion without requiring a central locus of control. We develop scalable speculation mechanisms including selective aborts of dependent tasks, which improves efficiency of speculative execution. Together, these enable Swarm to support an effective instruction window of hundreds of thousands of instructions across multiple processor cores.

2.4.2 Thread-Level Speculation

Thread-level speculation (TLS) allows multiple cores to execute tasks of a single sequential program in parallel while preserving the appearance of sequential execution. TLS techniques are focused on extracting fine-grained tasks from sequential programs. By operating on fine-grained tasks executed across multiple cores, TLS systems provide a larger effective instruction window compared to the instruction window on a single core [36].

In the TLS execution model [94, 194], a sequential instruction stream is divided at multiple points forming a sequence of tasks, where each task consists of contiguous in-

structions. Tasks are shipped to different cores, and executed speculatively and in parallel. Reads and writes from a task may have data dependences with other tasks. TLS systems ensure that such data dependences are handled properly such that the sequential semantics of the program are maintained. As tasks execute speculatively, the TLS system checks for cross-task data dependence violations or *conflicts*: a violation can occur when a read-after-write (RAW), write-after-write (WAW), or write-after-read (WAR) dependence executes out of order. When a conflict arises between two tasks, the one that comes later in program order is aborted and re-executed.

TLS allows a compiler to automatically parallelize portions of code in the presence of statically unknown data and control dependences. TLS extracts parallelism whenever dependences do not manifest at run-time. There are several approaches to delineate an instruction stream into tasks. The most common way is to split a sequential instruction stream at the beginning of each iteration of a loop or at the entry and exit points of a function call [159, 196].

Like most systems that support speculative parallelization, TLS systems implement three basic mechanisms: *conflict detection*, *version management* and *conflict resolution*.

Conflict detection: Most TLS schemes detect violations in a similar way: data that is speculatively accessed (e.g., read) is marked with some ordering tag, so that we can detect a later conflicting access (e.g., a write from another task) that should have preceded the first access in sequential order. Multiscalar processors use a centralized *address resolution buffer* (ARB) structure that observes all memory operations and detects ordering violations [79, 191]. However, this centralized approach does not scale well. Later proposals leverage caches and coherence protocols to detect conflicts [56, 194, 195]. They extend invalidation messages in the coherence protocol to carry the ordering tag corresponding to the task—dependence violations are flagged when a cache receives an invalidation message from a task with an earlier ordering tag. Most TLS systems perform conflict detection in an *eager* fashion, i.e., check for conflicts on each memory access during task execution. The alternative is to detect conflicts in a *lazy* manner at the end of task execution. However, this has limited benefits with strictly ordered fine-grained tasks.

Version management: A variety of approaches have been proposed to buffer and manage speculative state. In some proposals, tasks buffer unsafe state dynamically in caches [56, 78, 195], write buffers [94], or special buffers [79] to avoid corrupting main memory. In other proposals, tasks generate a log of updates that allow them to backtrack execution in case of a violation [85, 222]. Garzaran et al. [84] present a taxonomy of approaches to manage speculative memory state and the tradeoffs of the various approaches. Broadly, version management proposals may be categorized as *lazy* and *eager*.[2] In *lazy* version management, speculative data is buffered until a task is safe to commit. Garzaran et al.

[2]Garzaran et al. [84]'s taxonomy follows a different nomenclature—Future Main Memory (FMM) in their taxonomy corresponds to *eager* versioning nomenclature in this dissertation, while all other approaches fall under the *lazy* versioning nomenclature in this dissertation.

also distinguish *when* the speculative data is merged to memory on a task commit (either at task commit, or at some time after task commit time). By contrast, in *eager versioning*, speculative writes update memory directly and maintain an undo log to restore the original contents in case of an abort. Lazy versioning makes aborts fast since it just involves discarding the buffered speculative state. But commits are slow because they require merging of speculative state to main memory. By contrast, eager versioning makes commits fast as the data in memory is already updated. But it makes aborts slow, since it involves restoring the main memory to a correct state by walking the undo log. Most TLS proposals [56, 94, 191, 194] implement lazy versioning—for systems with small number of cores (<10 cores), these proposals find little difference between the two policies and favor lazy versioning owing to lower complexity in handling aborts.

Conflict resolution: On detecting a conflict, most TLS proposals [94, 191, 194] abort the task that caused the violation and all later speculative tasks because they may have consumed data written by the aborted task. Although more aggressive strategies are possible (such as squashing only the dependent later speculative tasks), they involve additional complexity in tracking and resolving reference chains among tasks.

Limitations of TLS: While TLS can extract fine-grained parallelism in applications, it suffers from two major drawbacks.

1. First, to maintain sequential semantics, the TLS execution model only allows spawning tasks in program order. However, we find that in many applications tasks are created widely out of order (Section 3.1). The control flow constructs of TLS schemes—loops and method calls—are insufficient to express order constraints among these tasks, instead necessitating software data structures to queue and schedule these tasks.
2. Second, TLS mechanisms often scale poorly beyond a few cores. Prior TLS schemes often abort all later speculative tasks to avoid tracking dependences among all tasks explicitly. While this is reasonable for small core counts, such unselective aborts are wasteful at larger core counts, limiting parallelism and scalability. Further, TLS schemes introduce a serialization bottleneck since they enforce in-order commits through token passing or by building successor lists [94, 171, 191, 195]. With fine-grained tasks, this bottleneck can cripple a system's performance and scalability at large core counts.

Addressing the limitations of TLS: Swarm (Chapter 3) addresses the limitations of TLS in the following ways:

1. Swarm presents an execution model based on timestamp-ordered tasks that conveys ordering constraints explicitly to hardware. New tasks are available to execute as soon as they are created, exposing more opportunities for parallelism. Hardware is responsible for queueing tasks and ensuring tasks commit in order.

2. Swarm employs a combination of eager-versioning and timestamp-based conflict detection to perform selective aborts of dependent tasks in a scalable manner. Further, Swarm achieves in-order commits without undue serialization by adapting techniques from distributed systems.

2.4.3 Transactional Memory

Transactional memory (TM) was proposed to simplify parallel programming, in particular the difficulty in managing shared state between threads [104]. TM allows programmers to declare certain code blocks as *transactions*. TM guarantees that user-defined transactions execute in an *atomic* and *isolated* manner with respect to other code blocks. Atomicity requires that all constituent actions in a transaction complete successfully, or that none of the actions appear to start executing. Isolation requires that transactions not interfere with each other, i.e., a transaction produces the same result as it would if no other transactions are executing concurrently. TM frees the programmer from the burden of orchestrating lock-based synchronization. Further, transactions are readily composable in contrast to lock-based programming.

TM systems execute multiple transactions in parallel with optimistic concurrency control as long as they do not conflict. Two transactions conflict if they access the same address and one of them writes. If they conflict, one of them is aborted and restarts. Similar to TLS, TM systems implement three basic mechanisms: *version management*, *conflict detection*, and *conflict resolution*. TM systems may implement these mechanisms in software or hardware or a combination of the two. Hardware transactional memory (HTM) implementations typically incur lower overheads compared to software transactional memory (STM) implementations.

			Versioning	
			Eager	Lazy
Conflict	**Detection**	Eager	LogTM [137], UTM [11], MetaTM [164]	VTM [163]
		Lazy	Impractical	TCC [95], FlexTM [184], BulkTM [42]

Table 2.1: HTM proposals for various points in the TM design space.

TM design space: Unlike TLS, TM systems do not need to support ordered execution of transactions; they only need to ensure a serializable schedule. This has enabled greater flexibility in the policies for the three basic mechanisms above. While TLS systems mostly employ lazy versioning and eager conflict detection, TM systems have explored other combinations as listed in Table 2.1. Similar to TLS, eager version management makes commits fast but aborts slow, while lazy version management offers complementary characteristics.

With eager conflict detection less work is potentially wasted if a transaction aborts; however, lazy conflict detection allows more transaction interleavings to successfully commit and guarantees forward progress. While TLS schemes resolve conflicts by aborting all later tasks, TM systems have choice in which transaction to abort leading to various conflict resolution proposals [11, 31, 42, 95, 111, 137]. Bobba et al. [32] present a taxonomy of the various policy choices in TM with their associated tradeoffs.

Improving implementation of speculation mechanisms: TM proposals also contribute several ideas to improving the implementation of the three basic mechanisms. Similar to TLS, many HTM systems implement data versioning and conflict detection by modifying the hardware caches and the coherence protocol. The earliest HTM proposal [104] duplicated the data cache, adding a separate *transactional cache* to track the reads and writes of a transaction. However, this incurs additional area and latency overhead. Later proposals [162, 204] instead extend the existing data cache lines with speculatively read (SR) and speculatively written (SW) bits, that are set when a transaction reads and writes a line respectively. Some proposals [197] also use dedicated buffers to track the read and write sets. A transaction detects a conflict when it receives a conflicting memory request from another core for a cache line in its read or write set (i.e., the SR or SW bit of the cache line is set). The underlying cache coherence protocol provides the mechanisms to communicate memory requests from different cores (or transactions).

LogTM [137] decouples version management from the data cache. It uses a per-thread log (allocated in virtual memory) to record older values of addresses being written by the transaction and writes the new value in place (eager version management). On a commit, this log is discarded and on an abort, a software handler executes, and restores the original values into memory by walking the log. LogTM also augments the coherence protocol to ensure other transactions do not observe speculative writes. LogTM makes eager version management and eager conflict detection practical, and achieves fast commits while making aborts slower. Bulk [42] decouples conflict detection from the data cache by recording the read and write sets in a hashed signature separate from the data cache. The signatures provide a compact representation of the address sets. While this reduces the state to track the address sets, it can introduce false positives (i.e., signal that a transaction has accessed a location, when in fact it has not), but there are no false negatives. LogTM-SE [219] combines the benefits of LogTM and Bulk to decouple both version management and conflict detection from the data cache.

Support for nesting: Transactional memory can support nesting to enable software composition. There are three types of nesting: *flat*, *closed*, and *open*. Flat nesting is the simplest type of nesting, and simply ignores the existence of nested transactions. Instead, all operations are executed as belonging to the parent transaction, resulting in large monolithic transactions. Aborting a nested transaction causes the parent transaction to be aborted. With closed nesting, the updates made by a nested transaction become externally visible only after the parent transaction commits [140]. When a nested transaction aborts,

the code in the nested transaction is re-executed without aborting the parent transaction, which often leads to improved performance. With open nesting, the updates of a nested transaction become externally visible as soon as it commits (even if the parent transaction has not committed yet) [6, 139]. When a parent transaction aborts, compensation actions should be executed to undo the effects of the nested transaction. In addition to the semantics of nesting, there are two design choices in how nested transactions execute. *Serial* nesting simply executes all nested transactions serially, while *parallel* nesting can launch multiple nested transactions on different threads to exploit nested parallelism.

Limitations of TM: TM systems suffer from the following limitations:

1. While support for TM enables efficient lock-free synchronization, TM does not provide support for order. The TM execution model is also restricted to spawning only as many transactions as the number of threads, which limits parallelism.
2. While TM can support nesting, many HTMs focus on parallelizing at the coarsest (shallowest) levels, incurring large overheads that squander the potential of nested parallelism. To guarantee atomicity, nested transactions merge their speculative state with their parent, and only the coarse, top-level transaction can commit. This results in large atomic blocks that are as expensive to track and as prone to abort as large serial transactions, adds substantial overheads, and suffers from subtle deadlock and livelock issues [20].
3. To perform conflict detection, many TM systems employ Bloom filters. However, these Bloom filters are allocated at the task granularity—two Bloom filters per task, one each to track the read and write sets respectively. Further, these Bloom filters are conservatively sized at design time. This leads to inefficient utilization for tiny transactions, while also offering no protection against high false positive rates for applications with very large transactions.

Addressing the limitations of TM: This dissertation addresses the limitations of TM in the following ways:

1. Swarm (Chapter 3) presents an execution model based on tasks with programmer-assigned timestamps to support order. Swarm also provides support for hardware queueing, thereby supporting many more tasks than number of threads or cores and exposing more parallelism. Swarm hardware builds on version management and conflict detection techniques developed for TM. We augment eager versioning with timestamp-based conflict detection— this supports ordered speculation and enables selective aborts which uncovers more parallelism. We also develop mechanisms to perform hierarchical conflict detection to reduce conflict checking traffic.
2. Fractal (Chapter 4) addresses the limitations of nesting in TM. Rather than building impossibly large atomic units, Fractal orders nested tasks to enforce atomicity. This allows Fractal to operate at the granularity of fine-grained tasks that are easy to track in hardware, while also avoiding subtle livelock and deadlock issues in nested

TM systems. Also, Fractal supports both ordered and unordered nested speculative parallelism unlike TM.

3. Amalgam (Chapter 5) matches the Bloom filter resources to the task size. Large address sets are tracked cooperatively by multiple Bloom filters. On the other hand, multiple tiny address sets are tracked in concert by a single Bloom filter when possible.

2.4.4 Fine-grained Messaging and Task Management

Managing fine-grained tasks in software incurs large overheads. Queue operations and synchronization through a shared-memory cache hierarchy incur significant latency (close to a hundred cycles [181]), thwarting the benefits of fine-grained parallelism, and limiting speedup. This is only exacerbated at larger core counts making fine-grained parallelism impractical. Prior work has shown that performing task management in hardware can yield significant benefits. Active messages lower the cost of sending tasks among cores [148, 210]. Carbon [120] introduces specialized hardware queues and a custom messaging protocol for enqueuing, dequeuing and distributing tasks across cores. Hardware also implements scheduling and load-balancing that operate concurrently with useful computation, without needing to communicate or synchronize via shared memory. Asynchronous direct messages [181] expose hardware primitives for sending and asynchronously receiving short messages between cores, enabling software to implement custom schedulers. Queueing operations do not rely on memory-mapped buffers, instead using low-latency instructions to send and receive task descriptors directly through registers. GPUs [215] and Anton 2 [90] feature custom schedulers for non-speculative tasks.

Swarm (Chapter 3) builds on these mechanisms, providing low-overhead asynchronous messaging for enqueueing and dequeuing tasks, and large hardware queues to manage fine-grained tasks. However, Swarm tasks are ordered and also run speculatively. Tasks are prioritized to run according to programmer-assigned timestamps. We manage hardware queue resources carefully, developing new mechanisms for queue virtualization and queue resource allocation, that ensure livelock and deadlock freedom. We also develop spatial hints (Section 3.7), a technique to exploit locality to achieve high-quality task mappings, and a novel load-balancer for speculative tasks (Section 3.10).

2.5 Bloom Filters

A Bloom filter [28] provides a means to compactly represent a set of n elements. Bloom filters are a popular choice for tracking address sets of tasks in systems that support speculative parallelization (such as hardware transactional memory) [42,135,180,219]. The filter is a bit-vector of m bits indexed by k independent hash functions, and supports operations such as element insertion and membership query.

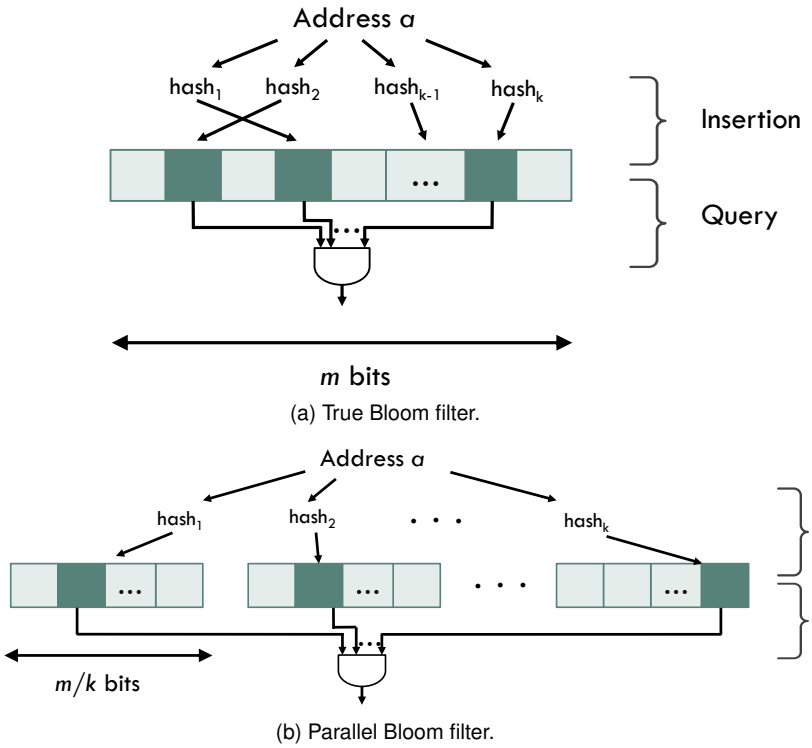


Figure 2-2: Bloom filter insertion and query for address a for (a) true Bloom filter and (b) parallel Bloom filter. The Bloom filter has m bits and k random hash functions.

Figure 2-2a illustrates the operation of a *true* Bloom filter. At initialization, every bit in the bit-vector is set to 0. To insert an address a to the set, we compute the k hash values of the address. Each hash function $h_i(a)$ returns a value in the range $[0, \cdots, m-1]$. The bits at the positions indicated by these values are set to 1. To test for membership of an address a, we compute the results of the k hash functions $h_i(a)$ and check the contents of the bits they point to. If at least one bit is set to 0, the address is not in the signature. If all the bits are set to 1, either the address is in the set or the insertion of other addresses set these bits to 1 (a false positive).

Implementing k hash function signatures using an SRAM with k read and write ports can be area inefficient (as the size of the SRAM cell increases quadratically with the number or ports) [180]. Hardware implementations of signatures using Bloom filters, instead use a *parallel* Bloom filter (Figure 2-2b). Instead of k hash functions, parallel Bloom filters have k Bloom filters each with $m/k-$bit-vectors, each with a different hash function. To insert an address we hash it to set a bit in each of the k filters. The test for membership is analogous, where we compute the results of the k hash functions and check the corresponding bits in each of the k Bloom filters. If at least one bit is 0, then the address is not in the set, else we have a match. This design uses smaller single-ported SRAMs instead of larger multi-ported SRAMS. The *parallel* Bloom filter can perform similar to a *true* Bloom filter but without the area inefficiency. We refer to the $m-$bit parallel Bloom filter comprising k single-hash Bloom filters of m/k bits each as a $m-$bit $k-$way Bloom filter.

False positives: Since Bloom filters encode a large space into a compact set, we can have false positives. For a $m-$bit $k-$way parallel Bloom filter, the probability of a false positive on a membership query, after n elements have been inserted is approximately given by $P_{FP}(n) = [1-(1-k/m)^n]^k$.[3] A larger bit-vector (m) decreases the false positive probability for a given number of insertions (n) independent of the other parameters of the Bloom filter. Further, as we move to smaller bit-vectors (m) the false positive probability grows rapidly; thus, in practice we require Bloom filters of a few hundred bits at least to keep false positive probabilities within acceptable bounds. While larger bit-vectors are preferable, they also increase the area needed for the filter. In general, a higher number of hash functions (i.e. larger k) is preferred since it reduces the false positive probability for small number of insertions (common case), while increasing the false positive probability at a large number of insertions.

Utility and limitations: Bloom filter membership queries are simple, requiring checking only a small number of bits (as many as the number of hash functions, k). Swarm (Chapter 3) leverages this property to develop new microarchitectural techniques that allow querying several Bloom filters in parallel using a single SRAM read. A key limitation of Bloom filters is that they often need to be a few hundred bits to keep false positive probabilities low. This can lead to inefficient utilization when tracking just one or two 64-bit addresses. Prior systems that utilize Bloom filters cannot address this inefficiency. Amalgam (Chapter 5) develops signature merging to combine multiple tiny address sets and track them using a single Bloom filter, thereby improving utilization.

[3] Christensen et al. [52] present an accurate analysis of the false positive probability for Bloom filters. We use the simpler, approximate equation here to more easily understand the effect of the different parameters on false positive probability.

3

Swarm: A Scalable Architecture for Ordered Parallelism

This work was conducted in collaboration with Mark C. Jeffrey, Cong Yan, Maleen Abeydeera, Joel Emer and Daniel Sanchez from MIT. The ideas for the distributed commit protocol, virtual-time based conflict detection, selective aborts, and spatial hints were developed collaboratively. This dissertation contributes hierarchical conflict detection using canaries, the hardware microarchitecture for efficient commit queue checks, and the load-balancing scheme. This dissertation also contributes to the development of applications and the simulator—in particular the memory hierarchy, conflict detection and correct handling of aborts, and event-driven simulation infrastructure.

PARALLELISM arises in a variety of flavors. Broadly, we can distinguish two classes of parallelism, *unordered* and *ordered*, that place different demands on the system. In unordered parallel programs, available tasks can execute and complete in any order. Tasks may have data dependences that are not known a priori. In this case, the programmer must use some form of *explicit synchronization*, such as locks or transactions, to arbitrate accesses to shared data. Unordered parallelism incurs small overheads in current multicores as long as tasks synchronize infrequently and are large enough to amortize the costs of software task management (e.g., scheduling and load balancing).

By contrast, ordered parallel programs consist of tasks that must follow a total or partial order. Tasks may have data dependences that are unknown a priori, but *synchronization is implicit*, determined by their order constraints. When tasks create new children tasks, they schedule them to run at a future time. Ordered parallelism is abundant in many domains, such as simulation, graph analytics, and databases. For example, consider a simulator for a parallel computer. Each task is an event (e.g., executing an instruction in a simulated core). Each task must run at a specific simulated time (introducing order constraints among tasks), and modifies a specific component (introducing data dependences among tasks). Tasks dynamically create other tasks (e.g., a simulated memory access), possibly for other components (e.g., a simulated cache), and schedule them for a future simulated time.

Prior work has tried to exploit ordered parallelism in software [100, 101], but has found that, in current multicores, runtime overheads negate the benefits of parallelism. This motivates the need for architectural support. Prior architectural techniques like thread-level speculation (TLS) [94, 171, 191, 195], which speculatively parallelizes sequential programs, is also unable to exploit ordered parallelism. This is due to two reasons (Section 3.2): First, many ordered algorithms have little parallelism when written as sequential programs. To enforce order constraints, sequential implementations introduce *false data dependences* among otherwise independent tasks. For example, a sequential timing simulator uses a priority queue to store future tasks, and priority queue accesses introduce false data dependences. Second, prior TLS schemes use techniques that scale poorly beyond few cores and cannot support large speculation windows.

We present *Swarm*, an architecture that exploits ordered parallelism efficiently. Swarm relies on a co-designed execution model and microarchitecture to scale. Swarm executes tasks speculatively and out of order, and speculates thousands of tasks ahead of the earliest active task to uncover enough parallelism. Swarm adapts existing version management and conflict detection schemes [219], and contributes new techniques that allow it to scale to large core counts and speculation windows. Specifically, we contribute the following novel techniques:

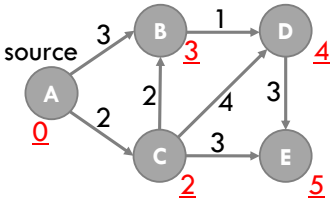
- An execution model based on tasks with programmer-specified timestamps that conveys order constraints to hardware without undue false data dependences.
- A hardware task management scheme that *(i)* features speculative task creation and dispatch, drastically reducing task management overheads, and *(ii)* implements a very large speculation window.
- A scalable conflict detection scheme that leverages eager versioning to, upon mispeculation, selectively abort the mispeculated task and its dependents.
- A distributed commit protocol that allows ordered commits without serialization, supporting multiple commits per cycle with modest communication.
- A technique called *spatial hints* that uses program knowledge to convey locality that *(i)* enables high-quality task mappings *(ii)* minimizes data movement.

We evaluate Swarm in simulation (Section 3.4 and Section 3.5) using seven challenging ordered applications: four graph analytics algorithms, two discrete-event simulators, and an in-memory database. Though they are not our focus, Swarm also scales unordered applications. At 64 cores, Swarm outperforms sequential implementations of these algorithms by 43–117×, and outperforms state-of-the-art parallel implementations on conventional multicores by 2.7–18.2×. At 256 cores, Swarm achieves a gmean speedup of 193× over the single-core performance across our benchmark suite.

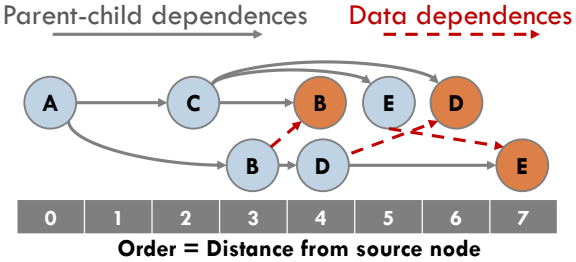
In summary, by making ordered execution scalable, Swarm speeds up challenging algorithms that are currently limited by stagnant single-core performance. Moreover, Swarm simplifies parallel programming, as it frees developers from using error-prone explicit synchronization.

```
prioQueue.enqueue(source, 0)
while prioQueue not empty:
 (node, dist) = prioQueue.dequeueMin()
 if node.distance not set:
  node.distance = dist
  for child in node.children:
   d = dist + distance(node, child)
   prioQueue.enqueue(child, d)
 else: // node already visited, skip
```

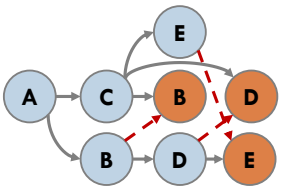
(a) Dijkstra's sssp code, highlighting the non-visited and visited paths that each task may follow



(b) Example graph and resulting shortest-path distances (underlined)



(c) Tasks executed by sssp. Each task shows the node it visits. Tasks that visit the same node have a data dependence



(d) A correct speculative schedule that achieves 2x parallelism

Figure 3-1: Dijkstra's single-source shortest paths algorithm (sssp) has plentiful ordered parallelism.

3.1 Motivation

3.1.1 Understanding Ordered Parallelism

Applications with ordered parallelism have three main characteristics [100, 155]. First, they consist of tasks that must follow a total or partial order. Second, tasks are not known in advance. Instead, tasks dynamically create children tasks, and schedule them to run at a future time. Task execution order is determined by the algorithm, and it is different from task creation order. Third, tasks may have data dependences that are not known a priori.

Ordered algorithms are common in many domains. First, they are common in graph analytics, especially in search problems [100, 155]. Second, they are important in simulating systems whose state evolves over time, such as circuits [136], computers [45, 169], networks [110, 208], health-care systems [115], and systems of partial differential equations [99, 127]. Third, they are needed in systems that must maintain externally-imposed order constraints, such as geo-replicated databases where transactions must appear to execute in timestamp order [57], or deterministic architectures [64, 130] and record-and-replay systems [108, 216] that constrain the schedule of parallel programs to ensure deterministic execution.

To illustrate the challenges in parallelizing these applications, consider Dijkstra's single-source shortest paths (`sssp`) algorithm [58, 80]. `sssp` finds the shortest distance between some source node and all other nodes in a graph with weighted edges. Figure 3-1(a) shows the sequential code for `sssp`, which uses a priority queue to store tasks. Each task operates on a single node, and is ordered by its tentative 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 all its children. Figure 3-1(b) shows an example graph, and Figure 3-1(c) shows the tasks that `sssp` executes to process this graph. Figure 3-1(c) shows the order of each task (its distance to the source node) in the *x*-axis, and outlines both parent-child relationships and data dependences. For example, task A at distance 0, denoted $(A, 0)$, creates children tasks $(C, 2)$ and $(B, 3)$; and tasks $(B, 3)$ and $(B, 4)$ both access node B, so they have a data dependence.

A distinctive feature of many programs with ordered parallelism is that task creation and execution order are different: children tasks are not immediately runnable, but are subject to a global order influenced by all other tasks in the program. For example, in Figure 3-1(c), $(C, 2)$ creates $(B, 4)$, but running $(B, 4)$ immediately would produce the wrong result, as $(B, 3)$, created by a different parent, must run first. Sequential implementations of these programs use scheduling data structures, such as priority or FIFO queues, to process tasks in the right order. These scheduling structures introduce false data dependences that restrict parallelism and hinder TLS (Section 3.2).

Order constraints limit non-speculative parallelism. For example, in Figure 3-1(c), only $(B, 4)$ and $(D, 4)$ can run in parallel without violating correctness. A more attractive option is to use speculation to elide order constraints. For example, Figure 3-1(d) shows a speculative schedule for `sssp` tasks. Tasks in the same *x*-axis position are executed simultaneously. This schedule achieves 2× parallelism in this small graph; larger graphs allow more parallelism (Section 3.1.2). This schedule produces the correct result because, although it elides order constraints, it happens to respect data dependences. Unfortunately, data dependences are not known in advance, so speculative execution must detect dependence violations and abort offending tasks to preserve correctness.

Recent work has tried to exploit ordered parallelism using speculative software runtimes [100, 101], but has found that the overheads of ordered, speculative execution negate the benefits of parallelism. This motivates the need for hardware support.

3.1.2 Analysis of Ordered Algorithms

To quantify the potential for hardware support and guide our design, we first analyze the parallelism and task structure of several ordered algorithms.

Benchmarks: We analyze six ordered benchmarks from the domains of graph analytics, simulation, and databases:

- `bfs` finds the breadth-first tree of an arbitrary graph.
- `sssp` is Dijkstra's algorithm (Section 3.1.1).

- `astar` uses the A$*$ pathfinding algorithm [97] to find the shortest route between two points in a road map.
- `msf` is Kruskal's minimum spanning forest algorithm [58].
- `des` is a discrete-event simulator for digital circuits. Each task represents a signal toggle at a gate input.
- `silo` is an in-memory OLTP database [205].

Section 3.4 describes their input sets and methodology details.

Analysis tool: We developed a pintool [131] to analyze these programs in x86-64. We focus on the instruction length, data read and written, and intrinsic data dependences of tasks, excluding the overheads and serialization introduced by the specific runtime used.

The tool uses a simple runtime that executes tasks sequentially. The tool profiles the number of instructions executed and addresses read and written (i.e., the read and write sets) of each task. It filters out reads and writes to the stack, the priority queue used to schedule tasks, and other run-time data structures such as the memory allocator. With this information, the tool finds the *critical path length* of the algorithm: the sequence of data-dependent tasks with the largest number of instructions. The tool then finds the *maximum achievable speedup* by dividing the sum of instructions of all tasks by the critical path length [217] (assuming unbounded cores and constant cycles per instruction). Note that this analysis *constrains parallelism only by true data dependences*: task order dictates the direction of data flow in a dependence, but is otherwise superfluous given perfect knowledge of data dependences.

Table 3.1 summarizes the results of this analysis. We derive three key insights that guide the design of Swarm:

Insight 1: Parallelism is plentiful. These applications have at least 158$\times$ maximum parallelism (`msf`), and up to 3440$\times$ (`bfs`). Thus, most order constraints are superfluous, making *speculative execution* attractive.

Insight 2: Tasks are small. Tasks are very short, ranging from a few tens of instructions (`bfs`, `sssp`, `msf`), to a few thousand (`silo`). Tasks are also relatively uniform: 90th-percentile instructions per task are close to the mean. Tasks have small read- and write-sets. For example, `sssp` tasks read 5.8 64-bit words on average, and write 0.4 words. Small tasks incur large overheads in software runtimes. Moreover, order constraints prevent runtimes from grouping tasks into coarser-grain units to amortize overheads. *Hardware support for task management* can drastically reduce these overheads.

Insight 3: Need a large speculation window. Table 3.1 also shows the achievable parallelism within a limited task window. With a T-task window, the tool does not schedule an independent task until all work more than T tasks behind has finished. Small windows

Application		bfs	sssp	astar	msf	des	silo
Maximum parallelism		3440×	793×	419×	158×	1440×	318×
Parallelism window=1K		827×	178×	62×	147×	198×	125×
Parallelism window=64		58×	26×	16×	49×	32×	17×
Instrs	mean	22	32	195	40	296	1969
	90th	47	70	508	40	338	2403
Reads	mean	4.0	5.8	22	7.1	50	88
	90th	8	11	51	7	57	110
Writes	mean	0.33	0.41	0.26	0.03	10.5	26
	90th	1	1	1	0	11	51
Max TLS parallelism		1.03×	1.10×	1.04×	158×	1.15×	45×

Table 3.1: Maximum achievable parallelism and task characteristics (instructions and 64-bit words read and written) of representative ordered applications.

severely limit parallelism. For example, parallelism in `sssp` drops from 793× with an infinite window, to 178× with a 1024-task window, to 26× with a 64-task window. Thus, for speculation to be effective, the architecture must support *many more speculative tasks than cores*.

These insights guide the design of Swarm. Our goal is to approach the maximum achievable parallelism while incurring only moderate overheads.

3.2 Background on HW Support for Speculative Parallelism

Much prior work has investigated thread-level speculation (TLS) schemes to parallelize sequential programs [84,94,170,171,191,196]. As discussed in Section 2.4.2, TLS schemes ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Although TLS schemes support ordered speculative execution, we find that two key problems prevent them from exploiting ordered parallelism:

1. False data dependences limit parallelism: To run under TLS, ordered algorithms must be expressed as sequential programs, but their sequential implementations have limited parallelism. Consider the code in Figure 3-1(a), where each iteration dequeues a task

from the priority queue and runs it, potentially enqueuing more tasks. Frequent data dependences *in the priority queue*, not among tasks themselves, cause frequent conflicts and aborts. For example, iterations that enqueue high-priority tasks often abort all future iterations.

Table 3.1 shows the maximum speedups that an ideal TLS scheme achieves on sequential implementations of these algorithms. These results use perfect speculation, an infinite task window, word-level conflict detection, immediate forwarding of speculative data, and no communication delays. Yet parallelism is meager in most cases. For example, `sssp` has 1.1× parallelism. Only `msf` and `silo` show notable speedups, because they need no queues: their task orders match loop iteration order.

The root problem is that loops and method calls, the control-flow constructs supported by TLS schemes, are insufficient to express the order constraints among these tasks. By contrast, Swarm implements a more general execution model with timestamp-ordered tasks to avoid software queues, and implements hardware priority queues integrated with speculation mechanisms, avoiding spurious aborts due to queue-related references.

2. Scalability bottlenecks: Although prior TLS schemes have developed scalable versioning and conflict detection schemes, two challenges limit their performance with large speculation windows and small tasks:

Forwarding vs selective aborts: Most TLS schemes find it is desirable to forward data written by an earlier, still-speculative task to later reader tasks. This prevents later tasks from reading stale data, reducing mispeculations on tight data dependences. However, it creates complex chains of dependences among speculative tasks. Thus, upon detecting mispeculation, most TLS schemes abort the task that caused the violation *and all later speculative tasks* [84, 94, 171, 191, 195]. TCC [95] and Bulk [42] are the exception: they do not forward data and only abort later readers when the earlier writer commits.

We find that forwarding speculative data is crucial for Swarm. However, while aborting all later tasks is reasonable with small speculative windows (2–16 tasks are typical in prior work), Swarm has a 4096-task window, and unselective aborts are impractical. To address this, we contribute a novel conflict detection scheme based on eager version management that allows both forwarding speculative data and selective aborts of dependent tasks.

Commit serialization: Prior TLS schemes enforce in-order commits by passing a token among ready-to-commit tasks [94, 171, 191, 195]. Each task can only commit when it has the token, and passes the token to its immediate successor when it finishes committing. This approach cannot scale to the commit throughput that Swarm needs. For example, with 100-cycle tasks, a 256-core system should commit 2.56 tasks/cycle on average. Even if commits were instantaneous, the latency incurred by passing the token makes this throughput unachievable.

To tackle this problem, we show that, by adapting techniques from distributed systems, we can achieve in-order commits without serialization, token-passing, or building successor lists.

3.3 Swarm: An Architecture for Ordered Parallelism

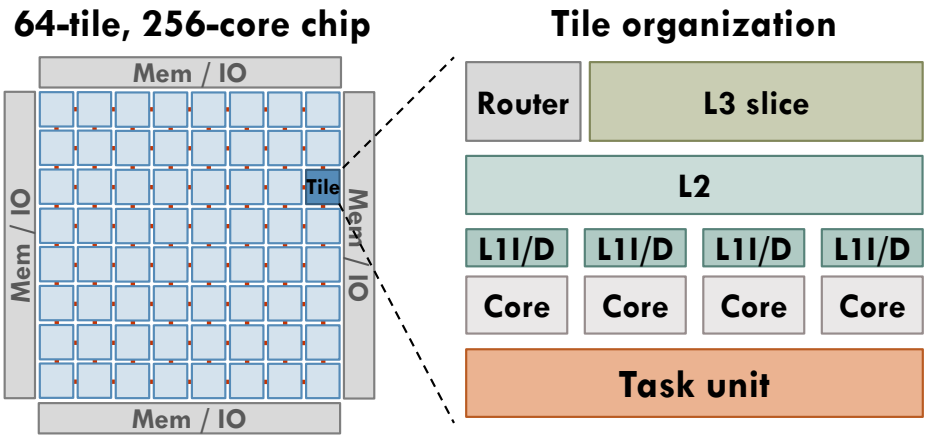


Figure 3-2: Swarm system and tile configuration.

Figure 3-2 shows Swarm's high-level organization. Swarm is a tiled, cache-coherent chip multiprocessor (CMP). Each tile has a group of simple cores. Each core has small, private, write-through L1 caches. All cores in a tile share a per-tile L2 cache, and each tile has a slice of a shared NUCA L3 cache. Each tile features a *task unit* that queues, dispatches, and commits tasks. Tiles communicate through a mesh NoC.

Key features: Swarm is optimized to execute short tasks with programmer-specified order constraints. Programmers define the execution order by assigning *timestamps* to tasks. Tasks can create children tasks with equal or later timestamps than their own. Tasks appear to execute in global timestamp order, but Swarm uses speculation to elide order constraints.

Swarm is coherently designed to support a large speculative task window efficiently. Swarm has no centralized structures: each tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Task units only communicate when they send new tasks to each other to maintain load balance, and, infrequently, to determine which finished tasks can be committed.

Swarm speculates far ahead of the earliest active task, and runs tasks even if their parent is still speculative. Figure 3-3(a) shows this process: a task with timestamp 0 is still running, but tasks with later timestamps and several speculative ancestors are running or have finished execution. For example, the task with timestamp 51, currently running, has three still-speculative ancestors, two of which have finished and are waiting to commit (8 and 20) and one that is still running (40).

Allowing tasks with speculative ancestors to execute uncovers significant parallelism, but may induce aborts that span multiple tasks. For example, in Figure 3-3(b) a new task with timestamp 35 conflicts with task 40, so 40 is aborted and child task 51 is both aborted and discarded. These aborts are *selective*, and only affect tasks whose speculative ancestors are aborted, or tasks that have read data written by an aborted task.

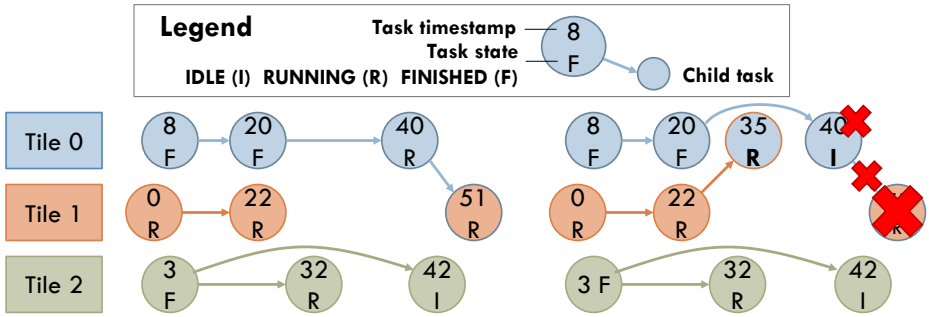


Figure 3-3: Example execution of `sssp`. By executing tasks even if their parents are speculative, SWARM uncovers ordered parallelism, but may trigger selective aborts.

We describe Swarm in a layered fashion. First, we present Swarm's execution model and ISA extensions. Second, we describe Swarm hardware assuming that *all queues are unbounded*. Third, we discuss how Swarm handles bounded queue sizes. Fourth, we present Swarm's hardware costs.

3.3.1 Swarm Execution Model

Swarm programs comprise timestamped tasks. Tasks appear to run in timestamp order. Tasks with the same timestamp may execute in any order, but run *atomically*—the system lazily selects an order for them.

A task can create one or more *children tasks* with an equal or later timestamp than its own. A child is ordered after its *parent*, but children with the same timestamp may execute in any order. Because hardware must track parent-child relations, tasks may create a limited number of children (8 in our implementation). Tasks that need more children enqueue a single task that creates them.

We expose these features using a low-level C++ API. Tasks are simply functions with signature:

```
void taskFn(timestamp, args...)
```

Tasks create children tasks by calling:

```
swarm::enqueue(taskFn, timestamp, [hint,] args...)
```

The hint is an optional argument that conveys locality information. Hints are described in detail in Section 3.7. If a task needs more than the maximum number of task descriptor arguments, three 64-bit words in our implementation, the runtime allocates them in memory.

3.3.2 ISA Extensions

Swarm manages and dispatches tasks using hardware task queues. A task is represented by a descriptor with the following architectural state: the task's function pointer, a 64-bit timestamp, and the task's arguments.

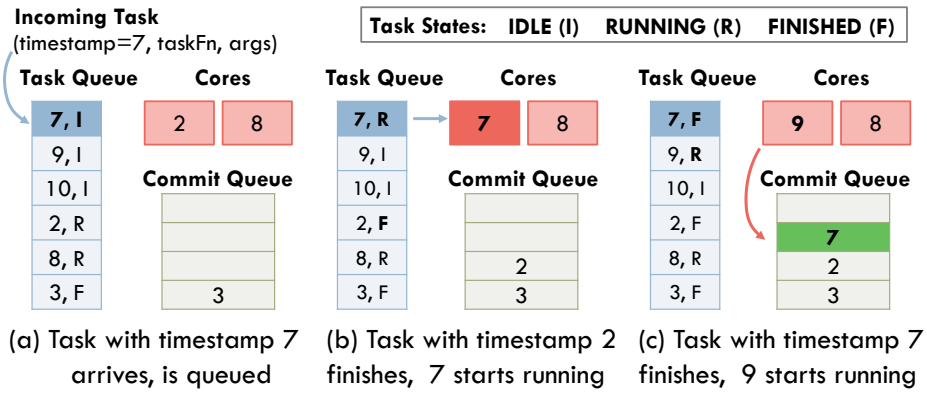


Figure 3-4: Task queue and commit queue utilization through a task's lifetime.

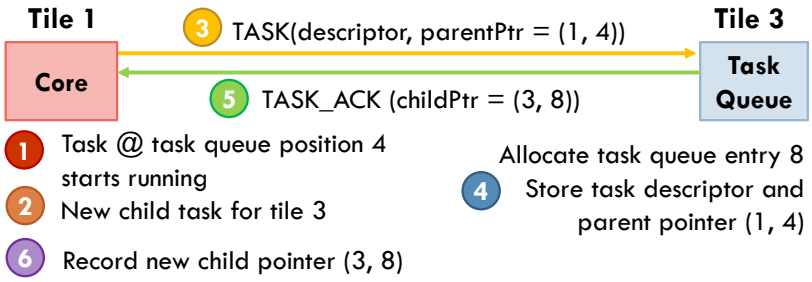


Figure 3-5: Task creation protocol. Cores send new tasks to other tiles for execution. To track parent-child relations, parent and child keep a pointer to each other.

Swarm adds instructions to enqueue and dequeue tasks. The `enqueue_task` instruction accepts a task descriptor (held in registers) as its input and queues the task for execution. A thread uses the `dequeue_task` instruction to start executing a previously-enqueued task. `dequeue_task` initiates speculative execution at the task's function pointer and makes the task's timestamp and arguments available (in registers). Task execution ends with a `finish_task` instruction.

`dequeue_task` stalls the core if an executable task is not immediately available, avoiding busy-waiting. When no tasks are left in any task unit and all threads are stalled on `dequeue_task`, the algorithm has terminated, and `dequeue_task` jumps to a configurable pointer to handle termination.

3.3.3 Task Queuing and Prioritization

The task unit has two main structures:

1. The *task queue* holds task descriptors (function pointer, timestamp, and arguments).
2. The *commit queue* holds the speculative state of tasks that have finished execution but cannot yet commit.

Figure 3-4 shows how these queues are used throughout the task's lifetime. Each new task allocates a task queue entry, and holds it until commit time. Each task allocates a commit

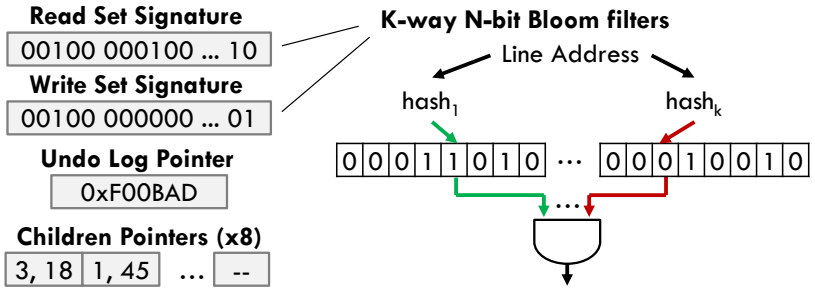


Figure 3-6: Speculative state for each task. Each core and commit queue entry maintains this state. Read and write sets are implemented with space-efficient Bloom filters.

queue entry when it finishes execution, and also deallocates it at commit time. For now, assume these queues always have free entries. Section 3.3.8 discusses what happens when they fill up.

Together, the task queue and commit queue are similar to a reorder buffer, but at task-level rather than instruction-level. They are separate structures because commit queue entries are larger than task queue entries, and typically fewer tasks are waiting to commit than to execute. However, unlike in a reorder buffer, tasks do not arrive in priority order. Both structures manage their free space with a freelist and allocate entries independently of task priority order, as shown in Figure 3-4.

Task enqueues: When a core creates a new task (through `enqueue_task`), it sends the task to a randomly-chosen target tile following the protocol in Figure 3-5. Parent and child track each other using *task pointers*. A task pointer is simply the tuple $(tile, task\ queue\ position)$. This tuple uniquely identifies a task because it stays in the same task queue position throughout its lifetime.

Task prioritization: Tasks are prioritized for execution in timestamp order. When a core calls `dequeue_task`, the highest-priority idle task is selected for execution. Since task queues do not hold tasks in priority order, an auxiliary *order queue* is used to find this task.

The order queue can be cheaply implemented with two small ternary content-addressable memories (TCAMs) with as many entries as the task queue (e.g., 256), each of which stores a 64-bit timestamp. With Panigrahy and Sharma's PIDR_OPT method [153], finding the next task to dispatch requires a single lookup in both TCAMs, and each insertion (task creation) and deletion (task commit or squash) requires two lookups in both TCAMs. SRAM-based implementations are also possible, but we find the small TCAMs to have a moderate cost (Section 3.3.9).

3.3.4 Speculative Execution and Versioning

The key requirements for speculative execution in Swarm are allowing fast commits and a large speculative window. To this end, we adopt *eager versioning*, storing speculative data

in place and logging old values. Eager versioning makes commits fast, but aborts are slow. However, Swarm's execution model makes conflicts rare, so eager versioning is the right tradeoff.

Eager versioning is common in hardware transactional memories [96, 137, 219], which do not perform ordered execution or speculative data forwarding. By contrast, most TLS systems use lazy versioning (buffering speculative data in caches) or more expensive multiversioning [42, 84, 94, 95, 157, 170, 171, 191, 195, 196] to limit the cost of aborts. Some early TLS schemes are eager [84, 222], and they still suffer from the limitations discussed in Section 3.2.

Swarm's speculative execution borrows from LogTM and LogTM-SE [137, 180, 219]. Our key contributions over these and other speculation schemes are *(i)* conflict detection (Section 3.3.5) and selective abort techniques (Section 3.3.6) that leverage Swarm's hierarchical memory system and Bloom filter signatures to scale to large speculative windows, and *(ii)* a technique that exploits Swarm's large commit queues to achieve high-throughput commits (Section 3.3.7).

Figure 3-6 shows the per-task state needed to support speculation: read- and write-set signatures, an undo log pointer, and child pointers. Each core and commit queue entry holds this state.

A successful `dequeue_task` instruction jumps to the task's code pointer and initiates speculation. Since speculation happens at the task level, there are no register checkpoints, unlike in HTM and TLS. Like in LogTM-SE, as the task executes, hardware automatically performs conflict detection on every read and write (Section 3.3.5). Then, it inserts the read and written addresses into the Bloom filters, and, for every write, it saves the old memory value in a memory-resident undo log. Stack addresses are not conflict-checked or logged.

When a task finishes execution, it allocates a commit queue entry; stores the read and write set signatures, undo log pointer, and children pointers there; and frees the core for another task.

3.3.5 Virtual Time-Based Conflict Detection

Conflict detection is based on a priority order that respects both programmer-assigned timestamps and parent-child relationships. Conflicts are detected at cache line granularity.

Unique virtual time: Tasks may have the same programmer-assigned timestamp. However, conflict detection has much simpler rules if tasks follow a total order. Therefore, tasks are assigned a *unique virtual time* when they are dequeued for execution. Unique virtual time is the 128-bit tuple *(programmer timestamp, dequeue cycle, tile id)*. The *(dequeue cycle, tile id)* pair is unique since at most one dequeue per cycle is permitted at a tile. Conflicts are resolved using this unique virtual time, which tasks preserve until they commit.

Unique virtual times incorporate the ordering needs of programmer-assigned timestamps and parent-child relations: children always start execution after their parents, so

a parent always has a smaller dequeue cycle than its child, and thus a smaller unique virtual time, even when parent and child have the same timestamp.

Conflicts and forwarding: Conflicts arise when a task accesses a line that was previously accessed by a later-virtual time task. Suppose two tasks, t_1 and t_2, are running or finished, and t_2 has a later virtual time. A read of t_1 to a line written by t_2 or a write to a line read or written by t_2 causes t_2 to abort. However, t_2 can access data written by t_1 even if t_1 is still speculative. Thanks to eager versioning, t_2 automatically uses the latest copy of the data—there is no need for speculative data forwarding logic [84].

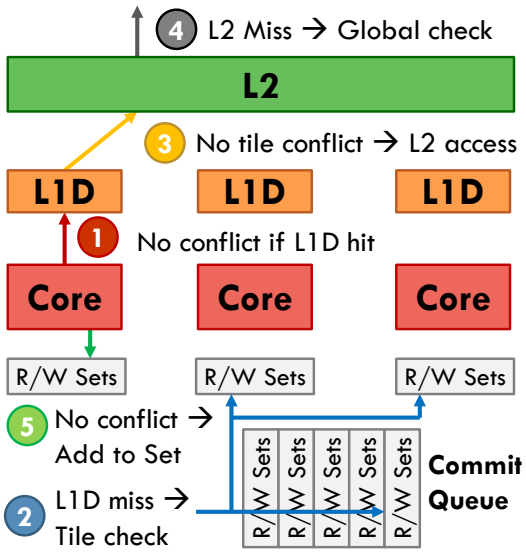


Figure 3-7: Local, tile, and global conflict detection for an access that misses in the L1 and L2.

Hierarchical conflict detection: Swarm exploits the cache hierarchy to reduce conflict checks. Figure 3-7 shows the different types of checks performed in an access:

1. The L1 is managed as described below to ensure L1 hits are conflict-free.
2. L1 misses are checked against other tasks in the tile (both in other cores and in the commit queue).
3. L2 misses, or L2 hits where a virtual time check (described below) fails, are checked against tasks in other tiles. As in LogTM [137], the L3 directory uses memory-backed *sticky bits* to only check tiles whose tasks may have accessed the line. Sticky bits are managed exactly as in LogTM.

Any of these conflicts trigger task aborts.

Using caches to filter checks: The key invariant that allows caches to filter checks is that, when a task with virtual time T installs a line in the (L1 or L2) cache, that line has no conflicts with tasks of virtual time $> T$. As long as the line stays cached with the right coherence permissions, it stays conflict-free. Because conflicts happen when tasks access

lines out of virtual time order, if another task with virtual time $U > T$ accesses the line, it is also guaranteed to have no conflicts.

However, accesses from a task with virtual time $U < T$ must trigger conflict checks, as another task with intermediate virtual time X, $U < X < T$, may have accessed the line. U's access does not conflict with T's, but may conflict with X's. For example, suppose a task with virtual time $X = 2$ writes line A. Then, task $T = 3$ in another core reads A. This is not a conflict with X's write, so A is installed in T's L1. The core then finishes T and dequeues a task $U = 1$ that reads A. Although A is in the L1, U has a conflict with X's write.

We handle this issue with two changes. First, when a core dequeues a task with a smaller virtual time than the one it just finished, it flushes the L1. Because L1s are small and write-through, this is fast, simply requiring to flash-clear the valid bits. Second, each L2 line has an associated *canary virtual time*, which stores the lowest task virtual time that need not perform a global check. For efficiency, lines in the same L2 set share the same canary virtual time. For simplicity, this is the maximum virtual time of the tasks that installed each of the lines in the set, and is updated every time a line is installed.

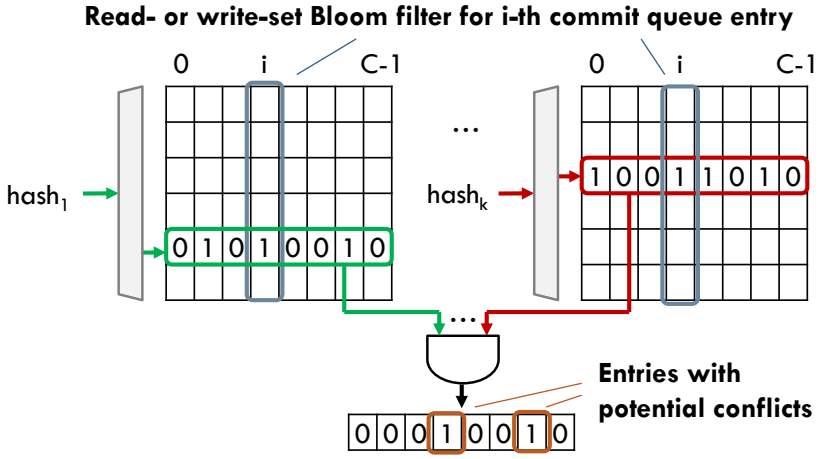


Figure 3-8: Commit queues store read- and write-set Bloom filters by columns, so a single access reads bit from all entries. All entries are checked in parallel.

Efficient commit queue checks: Although caches reduce the frequency of conflict checks, all tasks in the tile must be checked on every L2 access and on some global checks. To allow large commit queues (e.g., 64 tasks/queue), commit queue checks must be efficient. To this end, we leverage that checking a K-way Bloom filter only requires reading one bit from each way. As shown in Figure 3-8, Bloom filter ways are stored in columns, so a single 64-bit access per way reads all the necessary bits. Reading and ANDing all ways yields a word that indicates potential conflicts. For each queue entry whose position in this word is set, its virtual time is checked; those with virtual time higher than the issuing task's must be aborted.

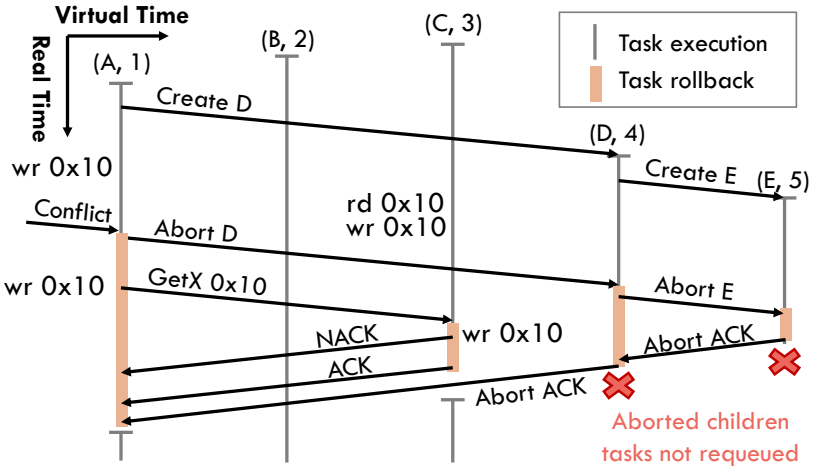


Figure 3-9: Selective abort protocol. Suppose $(A, 1)$ must abort after it writes 0x10. $(A, 1)$'s abort squashes child $(D, 4)$ and grandchild $(E, 5)$. During rollback, A also aborts $(C, 3)$, which read A's speculative write to 0x10. $(B, 2)$ is independent and thus not aborted.

3.3.6 Selective Aborts

Upon a conflict, Swarm aborts the later task and all its dependents: its children and other tasks that have accessed data written by the aborting task. Hardware aborts each task t in three steps:

1. Notify t's children to abort and be removed from their task queues.
2. Walk t's undo log in LIFO order, restoring old values. If one of these writes conflicts with a later-virtual time task, wait for it to abort and continue t's rollback.
3. Clear t's signatures and free its commit queue entry.

Applied recursively, this procedure selectively aborts all dependent tasks, as shown in Figure 3-9. This scheme has two key benefits. First, it reuses the conflict-detection logic used in normal operation. Undo-log writes (e.g., A's second `wr 0x10` in Figure 3-9) are normal conflict-checked writes, issued with the task's timestamp to detect all later readers and writers. Second, this scheme does not explicitly track data dependences among tasks. Instead, it uses the conflict-detection protocol to recover them as needed. This is important, because any task may have served speculative data to many other tasks, which would make explicit tracking expensive. For example, tracking all possible dependences on a 1024-task window using bit-vectors, as proposed in prior work [55, 161], would require $1024 \times 1023 \simeq 1$ Mbit of state.

3.3.7 Scalable Ordered Commits

To achieve high-throughput commits, Swarm adapts the virtual time algorithm [112], common in parallel discrete event simulation [76]. Figure 3-10 shows this protocol. Tiles periodically send the smallest unique virtual time of any unfinished (running or idle) task to an arbiter. Idle tasks do not yet have a unique virtual time and use *(timestamp, current*

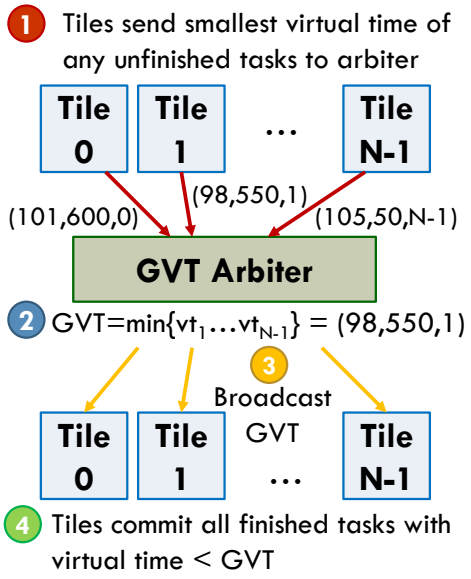


Figure 3-10: Global virtual time commit protocol.

cycle, tile id) for the purposes of this algorithm. The arbiter computes the minimum virtual time of all unfinished tasks, called the *global virtual time* (GVT), and broadcasts it to all tiles. To preserve ordering, only tasks with virtual time $<$ GVT can commit.

The key insight is that, by combining the virtual time algorithm with Swarm's large commit queues, *commit costs are amortized over many tasks*. A single GVT update often causes many finished tasks to commit. For example, if in Figure 3-10 the GVT jumps from (80,100,2) to (98,550,1), all tasks with virtual time (80,100,2)$< t <$(98,550,1) can commit. GVT updates happen sparingly (e.g., every 200 cycles) to limit bandwidth. Less frequent updates reduce bandwidth but increase commit queue occupancy.

In addition, eager versioning makes commits fast: a task commits by freeing its task and commit queue entries, a single-cycle operation. Thus, if a long-running task holds the GVT for some time, once it finishes, commit queues quickly drain and catch up to execution.

Compared with prior TLS schemes that use successor lists and token passing to reconcile order (Section 3.2), this scheme does not even require finding the successor and predecessor of each task, and does not serialize commits.

For the system sizes we evaluate, a single GVT arbiter suffices. Larger systems may need a hierarchy of arbiters.

3.3.8 Handling Limited Queue Sizes

The per-tile task and commit queues may fill up, requiring a few simple actions to ensure correct operation.

Task queue virtualization: Applications may create an unbounded number of tasks and schedule them for a future time. Swarm uses an overflow/underflow mechanism to give the illusion of unbounded hardware task queues [90, 120, 181]. When the per-tile task queue is nearly full, the task unit dispatches a special, non-speculative *coalescer* task to one of the cores. This coalescer task removes several *non-speculative*, idle task descriptors with high programmer-assigned timestamps from the task queue, stores them in memory, and enqueues a *splitter* task that will re-enqueue the overflowed tasks.

Note that only non-speculative task queue entries can be moved to software. These *(i)* are idle, and *(ii)* have no parent (i.e., their parent has already committed). When all entries are speculative, we need another approach.

Virtual time-based allocation: The task and commit queues may also fill up with speculative tasks. The general rule to avoid deadlock due to resource exhaustion is to always prioritize earlier-virtual time tasks, aborting other tasks with later virtual times if needed. For example, if a tile speculates far ahead, fills up its commit queue, and then receives a task that precedes all other speculative tasks, the tile must let the preceding task execute to avoid deadlock. This results in three specific policies for the commit queue, cores, and task queue.

Commit queue: If task t finishes execution, the commit queue is full, and t precedes any of the tasks in the commit queue, it aborts the highest-virtual time finished task and takes its commit queue entry. Otherwise, t stalls its core, waiting for an entry.

Cores: If task t arrives at the task queue, the commit queue is full, and t precedes all tasks in cores, t aborts the highest-virtual time task and takes its core.

Task queue: Suppose task t arrives at a task unit but the task queue is full. If some tasks are non-speculative, then a coalescer is running, so the task waits for a free entry. If all tasks in the task queue are speculative, the enqueued request is NACK'd (instead of ACK'd as in Figure 3-5) and the *parent task* stalls, and retries the enqueue using linear backoff. To avoid deadlock, we leverage that when a task's unique virtual time matches the GVT, it is the smallest-virtual time task in the system, and cannot be aborted. This task need not keep track of its children (no child pointers), and when those children are sent to another tile, they can be overflowed to memory if the task queue is full. This ensures that the GVT task makes progress, avoiding deadlock.

3.3.9 Analysis of Hardware Costs

We now describe Swarm's overheads. Swarm adds task units, a GVT arbiter, and modifies cores and L2s.

Table 3.2 shows the per-entry sizes, total queue sizes, and area estimates for the main task unit structures: task queue, commit queue, and order queue. All numbers are for one per-tile task unit. We assume a 16-tile, 64-core system as in Figure 3-2, with 256 task queue entries (64 per core) and 64 commit queue entries (16 per core). We use CACTI [202] for the task and commit queue SRAM areas (using 32 nm ITRS-HP logic) and scaled numbers

		Entries	Entry size	Size	Est. area
Task queue		256	51 B	12.75 KB	0.056 mm^2
Commit queue	filters	64	16×32 B	32 KB (2-port)	0.304 mm^2
	other	64	36 B	2.25 KB	0.012 mm^2
Order queue		256	2×8 B	4 KB (TCAM)	0.175 mm^2

Table 3.2: Sizes and estimated areas of main task unit structures.

from a commercial 28 nm TCAM [10] for the order queue area. Task queues use single-port SRAMs. Commit queues use several dual-port SRAMs for the Bloom filters (Figure 3-8), which are 2048-bit, 8-way in our implementation, and a single-port SRAM for all other state (unique virtual time, undo log pointer, and child pointers).

Overall, these structures consume 0.55 mm^2 per 4-core tile, or 8.8 mm^2 per chip, a minor cost. Enqueues and dequeues access the order queue TCAM, which consumes ∼70pJ per access [146]. Moreover, queue operations happen sparingly (e.g. with 100-cycle tasks, one enqueue and dequeue every 25 cycles), so energy costs are small.

The GVT arbiter is simple. It buffers a virtual time per tile, and periodically broadcasts the minimum one.

Cores are augmented with `enqueue/dequeue/finish_task` instructions (Section 3.3.2), the speculative state in Figure 3-6 (530 bytes), a 128-bit unique virtual time, and logic to insert addresses into Bloom filters and to, on each store, write the old value to an undo log. Finally, the L2 uses a 128-bit canary virtual time per set. For an 8-way cache with 64 B lines, this adds 2.6% extra state.

In summary, Swarm's costs are moderate, and, in return, confer significant speedups.

3.4 Experimental Methodology

Modeled system: We use a cycle-accurate, event-driven simulator based on Pin [131, 152] to model Swarm systems of up to 256 cores, as shown in Figure 3-2, with parameters in Table 3.3. We use detailed core, cache, network, and main memory models, and simulate all Swarm execution overheads (e.g., running mispeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). Our configuration is similar to the 256-core Kalray MPPA [62], though with a faster clock (the MPPA is a low-power part) and about 2× on-chip memory (the MPPA uses a relatively old 28 nm process).

We model in-order, single-issue cores. Cores run the x86-64 ISA. We use the decoder and functional-unit latencies of zsim's core model, which have been validated against Nehalem [178]. Cores are scoreboarded and stall-on-use, permitting multiple memory requests in flight.

Benchmarks: We use the six ordered benchmarks mentioned in Section 3.1.2: `bfs`, `sssp`, `astar`, `msf`, `des`, and `silo`. We port two other ordered benchmarks:

Cores	256 cores in 64 tiles (4 cores/tile), 2 GHz, x86-64 ISA; 8B-wide ifetch, 2-level bpred with 256×9-bit BHSRs + 512×2-bit PHT, single-issue, 4-entry ld/st buffers
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 8-way, inclusive, 7-cycle latency
L3 cache	64 MB, shared, static NUCA [117] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	16×16 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [213])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (16384 total), 16 commit queue entries/core (4096 total)
Swarm instrs	5 cycles per `enqueue/dequeue/finish_task`
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [40] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Commits	Tiles send updates to GVT arbiter every 200 cycles
Spills	Coalescers fire when a task queue is 85% full Coalescers spill up to 15 tasks each

Table 3.3: Configuration of the 256-core system.

- `color` uses the largest-degree-first heuristic [212] to assign distinct colors to adjacent graph vertices. This heuristic produces high-quality results and is thus most frequently used, but it is hard to parallelize.
- `nocsim` is a detailed network-on-chip simulator derived from GARNET [3]. Each task simulates an event at a component of a router.

We also port two unordered benchmarks from STAMP [134]:

- `genome` performs gene sequencing.
- `kmeans` implements K-means clustering.

We implement transactions with tasks of equal timestamp, so that they can commit in any order. As in prior work in transaction scheduling [14, 221] (Section 3.11.4), we break the original threaded code into tasks that can be scheduled asynchronously and generate children tasks as they find more work to do. Table 3.4 details the applications' provenance and input sets.

For most benchmarks, we use tuned serial and state-of-the-art parallel versions from existing suites (Table 3.4). We then port each serial implementation to Swarm. Swarm versions use fine-grained tasks, but use the same data structures and perform the same work as the serial version, so differences between serial and Swarm versions stem from parallelism, not other optimizations.

	Source	Input	Swarm 1-core	
			Run-time	**Perf vs serial**
bfs	PBFS [124]	hugetric-00020 [18, 61]	3.59 Bcycles	−18%
sssp	Galois [155]	East USA roads [1]	3.21 Bcycles	+33%
astar	Own	Germany roads [151]	1.97 Bcycles	+1%
color	[98]	com-youtube [126]	1.65 Bcycles	+54%
des	Galois [155]	csaArray32	1.92 Bcycles	+70%
nocsim	GARNET [3]	16x16 mesh, tornado traffic	22.37 Bcycles	+68%
silo	[205]	TPC-C, 4 whs, 32 Ktxns	2.83 Bcycles	+16%
msf	PBBS [186]	kronecker_logn16 [18, 61]	2.16 Bcycles	+2%
genome	STAMP [134]	-g4096 -s48 -n1048576	2.30 Bcycles	+1%
kmeans	STAMP [134]	-m40 -n40 -i rnd-n16K-d24-c16	8.56 Bcycles	+2%

Table 3.4: Benchmark information: source implementations, inputs, run-times on a 1-core Swarm system, 1-core speedups over tuned serial implementations.

We wrote our own tuned serial and Swarm `astar` implementations. `astar` is notoriously difficult to parallelize—to scale, prior work in parallel pathfinding sacrifices solution quality for speed [35]. Thus, we do not have a software-only parallel implementation.

We port `silo` to show that Swarm can extract ordered parallelism from applications that are typically considered unordered. Database transactions are unordered in `silo`. We decompose each transaction into many small ordered tasks to exploit *intra-transaction parallelism*. Tasks from different transactions use disjoint timestamp ranges to preserve atomicity. This exposes significant fine-grained parallelism within and across transactions.

Input sets: We use a varied set of inputs, often from standard collections such as DIMACS (Table 3.4). `bfs` operates on an unstructured mesh; `sssp` and `astar` use large road maps; `msf` uses a Kronecker graph; `des` simulates an array of carry-select adders; and `silo` runs the TPC-C benchmark on 4 warehouses. `color` operates on a YouTube social graph [126]. `nocsim` simulates a 16x16 mesh with tornado traffic at a per-tile injection rate of 0.06. STAMP benchmarks use inputs between the recommended "+" and "++" sizes, to achieve a run time large enough to evaluate 256-core systems, yet small enough to be simulated in reasonable time. We fix the number of `kmeans` iterations to 40 for consistency across runs.

All benchmarks have serial run-times of over one billion cycles (Table 3.4). We have evaluated other inputs (e.g., random and scale-free graphs), and qualitative differences are not affected. Note that some inputs can offer plentiful trivial parallelism to a software algorithm. For example, on large, shallow graphs (e.g., 10 M nodes and 10 levels), a simple bulk-synchronous `bfs` that operates on one level at a time scales well [124]. But we use a graph with 7.1 M nodes and 2799 levels, so `bfs` must speculate across levels to uncover enough parallelism.

For each benchmark, we fast-forward to the start of the parallel region (skipping initialization), and report results for the full parallel region. We perform enough runs to achieve 95% confidence intervals $\leq 1\%$.

Idealized memory allocation: Dynamic memory allocation is not simulated in detail, and a scalable solution is left to future work. Only `des`, `nocsim` and `silo` tasks allocate memory frequently, and data dependences in the system's memory allocator serialize them. In principle, we could build a task-aware allocator with per-core memory pools to avoid serialization. However, building high-performance allocators is complex [86,183]. Instead, the simulator allocates and frees memory in a task-aware way. Freed memory is not reused until the freeing task commits to avoid spurious dependences. Each allocator operation incurs a 30-cycle cost. For fairness, *serial and software-parallel implementations also use this allocator*. We believe this simplification will not significantly affect `des`, `nocsim` and `silo` results when simulated in detail.

3.5 Evaluation of Swarm

We evaluate Swarm systems of 1- to 64-cores here using the six ordered benchmarks described in Section 3.1.2. We introduce mechanisms that enable Swarm to exploit locality in order to scale up to 256-cores in Section 3.7, and evaluate the full suite of applications then. We first compare Swarm with alternative implementations, then analyze its behavior in depth.

3.5.1 Swarm Scalability

Figure 3-11 shows Swarm's performance on 1- to 64-core systems. In this experiment, *per-core* queue and L2/L3 capacities are kept constant as the system grows, so *systems with more cores have higher queue and cache capacities*. This captures performance per unit area.

Each line in Figure 3-11 shows the speedup of a single application over a 1-core system (i.e., its self-relative speedup). At 64 cores, speedups range from $51\times$ (`msf`) to $122\times$ (`sssp`), demonstrating high scalability. In addition to parallelism, the larger queues and L3 of larger systems also affect performance, causing super-linear speedups in some benchmarks (`sssp`, `bfs`, and `astar`). We tease apart the contribution of these factors in Section 3.5.3.

3.5.2 Swarm vs Software Implementations

Figure 3-12 compares the performance of the Swarm and software-only versions of each benchmark. Each graph shows the speedup of the Swarm and software-parallel versions over the tuned serial version running on a system of the same size, from 1 to 64 cores. As in Figure 3-11, queue and L2/L3 capacities scale with the number of cores.

Swarm outperforms the serial versions by 43–$117\times$, and the software-parallel versions by 2.7–$18.2\times$. We analyze the reasons for these speedups for each application.

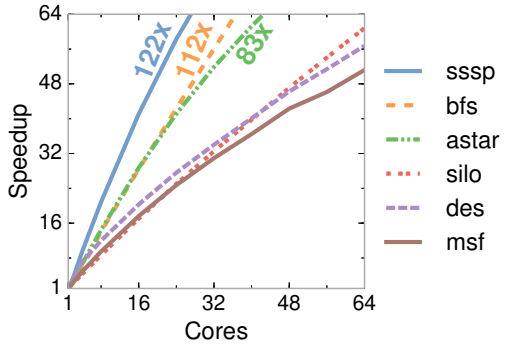


Figure 3-11: SWARM self-relative speedups on 1-64 cores. Larger systems have larger queues and caches, which affect speedups and sometimes cause superlinear scaling.

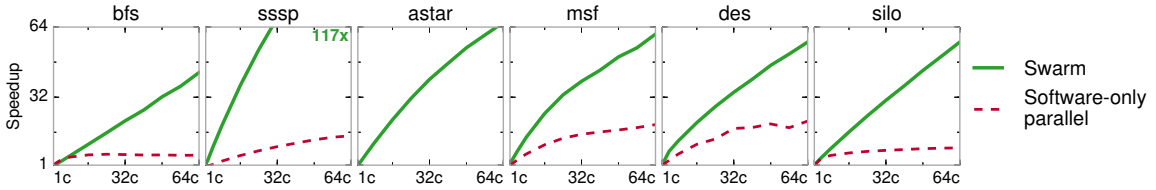


Figure 3-12: Speedup of SWARM and state-of-the-art software-parallel implementations from 1 to 64 cores, relative to a tuned serial implementation running on a system of the same size.

bfs: Serial `bfs` does not need a priority queue. It uses an efficient FIFO queue to store the set of nodes to visit. At 1 core, Swarm is 33% *slower* than serial `bfs`; however, Swarm scales to 43× at 64 cores. By contrast, the software-parallel version, PBFS [124], scales to 6.0×, then slows down beyond 24 cores. PBFS only works on a single level of the graph at a time, while Swarm speculates across multiple levels.

sssp: Serial `sssp` uses a priority queue. Swarm is 32% faster at one core, and 117× faster at 64 cores. The software-parallel version uses the Bellman-Ford algorithm [58]. Bellman-Ford visits nodes out of order to increase parallelism, but wastes work in doing so. Threads in Bellman-Ford communicate infrequently to limit overheads [100], wasting much more work than Swarm's speculative execution. As a result, Bellman-Ford `sssp` scales to 14× at 64 cores, 8.1× slower than Swarm.

astar: Our tuned serial `astar` uses a priority queue to store tasks [58]. Swarm outperforms it by 2% at one core, and by 66× at 64 cores.

msf: The serial and software-parallel `msf` versions sort edges by weight to process them in order. Our Swarm implementation instead does this sort implicitly through the task queues, enqueuing one task per edge and using its weight as the timestamp. This allows Swarm to overlap the sort and edge-processing phases. Swarm outperforms the serial version by 70% at one core and 61× at 64 cores. The software-parallel `msf` uses software

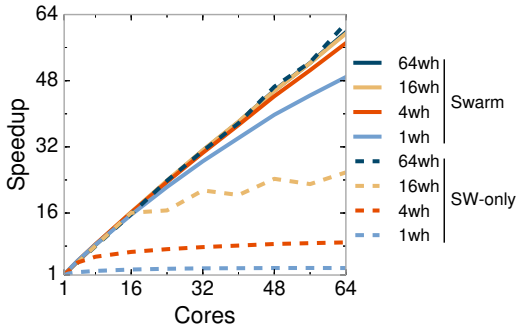


Figure 3-13: Speedup of SWARM and software `silo` with 64, 16, 4, and 1 TPC-C warehouses.

speculation via deterministic reservations [25], and scales to 19× at 64 cores, 3.1× slower than Swarm.

des: Serial `des` uses a priority queue to simulate events in time order. Swarm outperforms the serial version by 23% at one core, and by 57× at 64 cores. The software-parallel version uses the Chandy-Misra-Bryant (CMB) algorithm [136, 192]. CMB exploits the simulated communication latencies among components to safely execute some events out of order (e.g., if two nodes have a 10-cycle simulated latency, they can be simulated up to 9 cycles away). CMB scales to 21× at 64 cores, 2.7× slower than Swarm. Half of Swarm's speedup comes from exploiting speculative parallelism, and the other half from reducing overheads.

silo: Serial `silo` runs database transactions sequentially without synchronization. Swarm outperforms serial `silo` by 10% at one core, and by 57× at 64 cores. The software-parallel version uses a carefully optimized protocol to achieve high transaction rates [205]. Software-parallel `silo` scales to 8.8× at 64 threads, 6.4× slower than Swarm. The reason is fine-grained parallelism: in Swarm, each task reads or writes at most one tuple. This exposes parallelism within and across database transactions, and reduces the penalty of conflicts, as only small, dependent tasks are aborted instead of full transactions.

Swarm's benefits on `silo` heavily depend on the amount of coarse-grained parallelism, which is mainly determined by the number of TPC-C warehouses. To quantify this effect, Figure 3-13 shows the speedups of Swarm and software-parallel `silo` with 64, 16, 4, and 1 warehouses. With 64 warehouses, software-parallel `silo` scales linearly up to 64 cores and is 4% faster than Swarm. With fewer warehouses, database transactions abort frequently, limiting scalability. With a single warehouse, software-parallel `silo` scales to only 2.7×. By contrast, Swarm exploits fine-grained parallelism within each transaction, and scales well even with a single warehouse, by 49× at 64 cores, 18.2× faster than software-parallel `silo`.

Overall, these results show that Swarm outperforms a wide range of parallel algorithms, even when they use application-specific optimizations. Moreover, Swarm implementations use no explicit synchronization and are simpler, which is itself valuable.

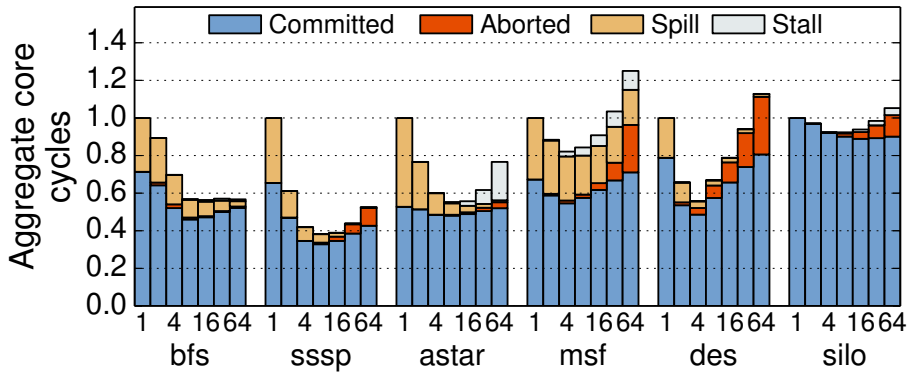


Figure 3-14: Breakdown of total core cycles for SWARM systems with 1 to 64 cores. Most time is spent executing tasks that are ultimately committed.

3.5.3 Swarm Analysis

We now analyze the behavior of different benchmarks in more detail to gain insights about Swarm.

Cycle breakdowns: Figure 3-14 shows the breakdown of aggregate core cycles. Each set of bars shows results for a single application as the system scales from 1 to 64 cores. The height of each bar is the sum of cycles spent by all cores, normalized by the cycles of the 1-core system (lower is better). With linear scaling, all bars would have a height of 1.0; higher and lower bars indicate sub- and super-linear scaling, respectively. Each bar shows the breakdown of cycles spent executing tasks that are ultimately committed, tasks that are later aborted, spilling tasks from the hardware task queue (using coalescer and splitter tasks, Section 3.3.8), and stalled.

Swarm spends most of the cycles executing tasks that later commit. At 64 cores, aborted work ranges from 1% (`bfs`) to 27% (`des`) of cycles. All graph benchmarks spend significant time spilling tasks to memory, especially with few cores (e.g., 47% of cycles for single-core `astar`). In all benchmarks but `msf`, spill overheads shrink as the system grows and task queue capacity increases; `msf` enqueues millions of edges consecutively, so larger task queues do not reduce spills. Finally, cores rarely stall due to full or empty queues. Only `astar` and `msf` spend more than 5% of cycles stalled at 64 cores: 27% and 8%, respectively.

Figure 3-14 also shows the factors that contribute to super-linear scaling in Figure 3-11. First, larger task queues can capture a higher fraction of runnable tasks, reducing spills. Second, larger caches can better fit the working set, reducing the cycles spent executing committed tasks (e.g., `silo`). However, beyond 4–8 cores, the longer hit latency of the larger NUCA L3 counters its higher hit rate in most cases, increasing execution cycles.

Speedups with idealizations: To factor out the impact of queues and memory system on scalability, we consider systems with two idealizations: unbounded queues, which fac-

Speedups	1c vs 1c-base	64c vs 1c-base	64c vs 1c
SWARM baseline	1×	77×	77×
+ unbounded queues	1.4×	87×	61×
+ 0-cycle mem system	5×	274×	54×

Table 3.5: gmean speedups with progressive idealizations: unbounded queues and a zero-cycle memory system (1c-base = 1-core SWARM baseline without idealizations).

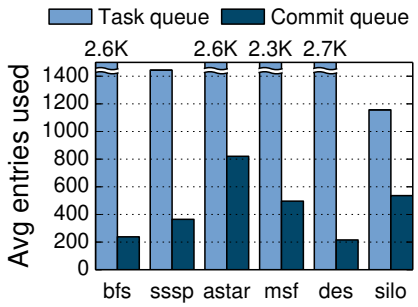


Figure 3-15: Average task and commit queue occupancies for 64-core SWARM.

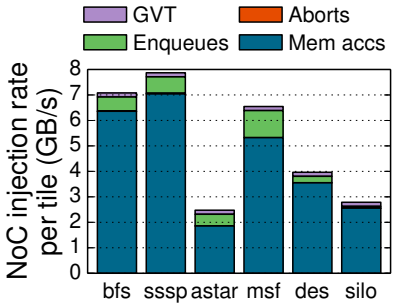


Figure 3-16: Breakdown of NoC traffic per tile for 64-core, 16-tile SWARM.

tor out task spills, and an ideal memory system with 0-cycle delays for all accesses and messages. Table 3.5 shows the gmean speedups when these idealizations are progressively applied. The left and middle columns show 1- and 64-core speedups, respectively, over the 1-core baseline (without idealizations). While idealizations help both cases, they have a larger impact on the 1-core system. Therefore, the 64-core speedups relative to the 1-core system *with the same idealizations* (right column) are lower. With all idealizations, this speedup is purely due to exploiting parallelism; 64-core Swarm is able to mine 54× parallelism on average (46×–63×).

Queue occupancies: Figure 3-15 shows the average number of task queue and commit queue entries used across the 64-core system. Both queues are often highly utilized. Commit queues can hold up to 1024 finished tasks (64 per tile). On average, they hold from 216 in `des` to 821 in `astar`. This shows that cores often execute tasks out of order, and these tasks wait a significant time until they commit—a large speculative window is crucial, as the analysis in Section 3.1.2 showed. The 4096-entry task queues are also well utilized, with average occupancies between 1157 (`silo`) and 2712 (`msf`) entries.

Network traffic breakdown: Figure 3-16 shows the NoC traffic breakdown at 64 cores (16 tiles). The cumulative injection rate per tile remains well below the saturation injection rate (32 GB/s). Each bar shows the contributions of memory accesses (between the L2s and L3) issued during normal execution, tasks enqueues to other tiles, abort traffic (including child abort messages and rollback memory accesses), and GVT updates. Task

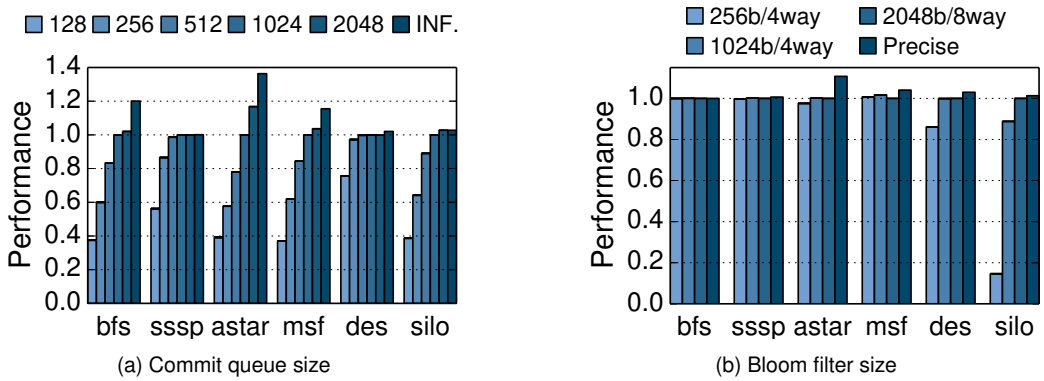


Figure 3-17: Sensitivity of 64-core SWARM to commit queue and Bloom filter sizes.

enqueues, aborts, and GVT updates increase network traffic by 15% on average. Thus, Swarm imposes small overheads on traffic and communication energy.

Conflict detection energy: Conflict detection requires Bloom filter checks—performed in parallel over commit queue entries (Figure 3-7)—and for those entries where the Bloom filter reports a match, a virtual time check to see whether the task needs to be aborted. Both events happen relatively rarely. Each tile performs one Bloom filter check every 8.0 cycles on average (from 2.5 cycles in `msf` to 13 cycles in `bfs`). Each tile performs one timestamp check every 49 cycles on average (from 6 cycles in `msf` to 143 cycles in `astar`). Hence, Swarm's conflict detection imposes acceptable energy overheads.

Canary virtual times: To lower overheads, all lines in the same L2 set share a common canary virtual time. This causes some unnecessary global conflict checks, but we find the falsely unfiltered checks are infrequent. At 64 cores, using precise per-line canary virtual times reduces global conflict checks by 10.3% on average, and improves application performance by less than 1%.

3.5.4 Sensitivity Studies

We explore Swarm's sensitivity to several design parameters at 64 cores:

Commit queue size: Figure 3-17a shows the speedups of different applications as we sweep aggregate commit queue entries from 128 (8 tasks per tile) to unbounded; the default is 1024 entries. Commit queues are fundamental to performance: fewer than 512 entries degrade performance considerably. More than 1024 entries confer moderate performance boosts to some applications. We conclude that 1024 entries strikes a good balance between performance and implementation cost for the benchmarks we study.

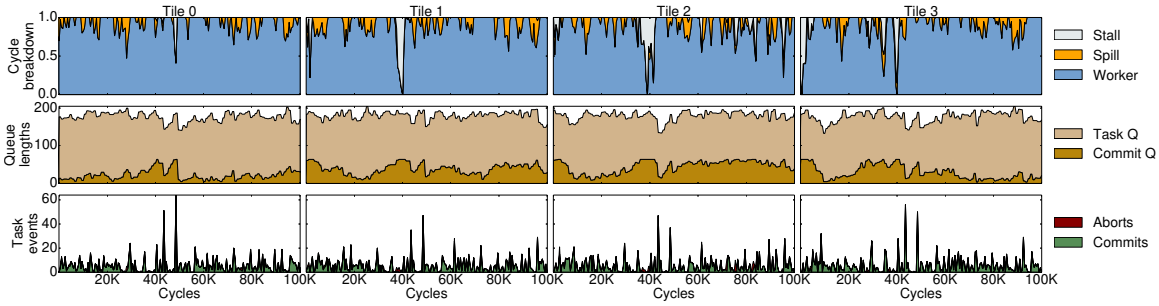


Figure 3-18: Execution trace of `astar` on 16-core (4-tile) SWARM over a 100 Kcycle interval: breakdown of core cycles (top), queue lengths (middle), and task commits and aborts (bottom) for each tile.

Bloom filter configuration: Figure 3-17b shows the relative performance of different Bloom filter configurations. The default 2048-bit 8-way Bloom filters achieve performance within 10% of perfect conflict detection. Smaller Bloom filters cause frequent false positives and aborts in `silo` and `des`, which have the tasks with the largest footprint. However, `bfs`, `sssp`, and `msf` tasks access little data, so they are insensitive to Bloom filter size.

Frequency of GVT updates: Swarm is barely sensitive to the frequency of GVT updates. As we vary the period between GVT updates from 50 cycles to 800 cycles (the default is 200 cycles), performance at 64 cores drops from 0.1% in `sssp` to 3.0% in `msf`.

3.5.5 Swarm Case Study: `astar`

Finally, we present a case study of `astar` running on a 16-core, 4-tile system to analyze Swarm's time-varying behavior. Figure 3-18 depicts several per-tile metrics, sampled every 500 cycles, over a 100 Kcycle interval: the breakdown of core cycles (top row), commit and task queue lengths (middle row), and tasks commit and abort events (bottom row). Each column shows these metrics for a single tile.

Figure 3-18 shows that task queues are highly utilized throughout the interval. As task queues approach their capacity, coalescer tasks kick in, spilling tasks to memory. Commit queues, however, show varied occupancy. As tasks are executed out of order, they use a commit queue entry until they are safe to commit (or are aborted). Most of the time, commit queues are large enough to decouple execution and commit orders, and tiles spend the vast majority of time executing worker tasks.

Occasionally, however, commit queues fill up and cause the cores to stall. For example, tiles stall around the 40 Kcycle mark as they wait for a few straggler tasks to finish. The last of those stragglers finishes at 43 Kcycles, and the subsequent GVT update commits a large number of erstwhile speculative tasks, freeing up substantial commit queue space. These events explain `astar`'s sensitivity to commit queue size as seen in Figure 3-17a.

Finally, note that although queues fill up rarely, commits tend to happen in bursts throughout the run. This shows that fast commits are important, as they enable Swarm to quickly turn around commit queue entries.

3.6 Scaling Swarm to Larger System Sizes: Need For Spatial Mapping

By default, Swarm sends new tasks to a random tile. While this is good for load balance, it leads to increased data movement costs as we scale to larger system sizes, and consequently reduced performance. To resolve this, we develop *spatial hints*, a technique that uses program knowledge to achieve high-quality task mappings that minimizes data movement. At first glance, it might seem that spatial mapping and speculation are at odds: achieving a good spatial mapping requires knowing the data accessed by each task, but the key advantage of speculation is precisely that one need not know the data accessed by each task. However, we find that *there is a wide gray area*: in many applications, *most* of the data accessed is known at run-time when the task is created. Thus, there is ample information to achieve high-quality spatial task mappings. Beyond reducing data movement, high-quality mappings also enhance parallelism by making most conflicts local. Finally, while hints improve locality and reduce conflicts, they can also cause load imbalance. We thus design a load balancer that leverages hints to redistribute tasks across tiles in a locality-aware fashion.

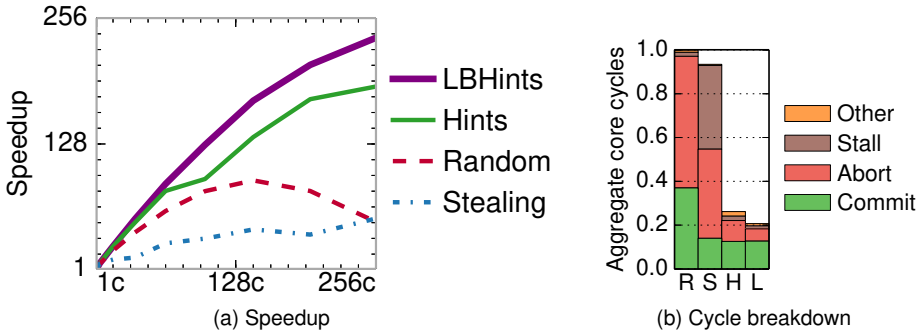


Figure 3-19: Performance of Random, Stealing, Hints, and LBHints schedulers on `des`: (a) speedup relative to 1-core Swarm, and (b) breakdown of total core cycles at 256 cores, relative to Random.

To explore the impact of spatial task mapping, we compare the performance of the `des` benchmark under different schedulers. We simulate systems of up to 256 cores as shown in Figure 3-2, with parameters as in Table 3.3. We compare four schedulers:

- Random is Swarm's default task mapping strategy. New tasks are sent to a random tile for load balance.
- Stealing is an *idealized work-stealing* scheduler, the most common scheduler in non-speculative programs [2, 29]. New tasks are enqueued to the local tile, and tiles that run out of tasks steal tasks from a victim tile. To evaluate stealing in the best possible light, we do not simulate any stealing overheads: out-of-work tiles instantaneously find the tile with the most idle tasks and steal the earliest-timestamp task. Work-stealing is sensitive to the stealing policies used (Section 3.11.5). We studied a wide range of policies, both in terms of victim tile selection (random, nearest-

neighbor, most-loaded) and task selection within a victim tile (earliest-timestamp, random, latest-timestamp) and empirically found the selected policies to perform best overall for our benchmarks.

- Hints is our hint-based spatial task mapping scheme.
- LBHints is our hint-based load balancer.

In des, Hints maps each gate in the simulated circuit to a specific, statically chosen tile. New tasks are sent to the tile where the gate they operate on is mapped (Section 3.7). LBHints enhances Hints by periodically remapping gates across tiles to equalize their load (Section 3.10).

Two factors make spatial mapping in des possible. First, each task operates on a single gate. Second, this gate is known at run-time when the task is created. As we will see later, good spatial mappings are possible even when these conditions are not completely met (i.e., tasks access multiple pieces of data, or some of the data they access is not known at task creation time). Also, note that even when we know all data accesses, speculation is still needed, as tasks can be created out of order and executed in the wrong order.

Figure 3-19a compares the performance of different schemes on 1- to 256-core systems. Each line shows the speedup relative to a 1-core Swarm system (all schedulers are equivalent at 1 core). Stealing performs worst, scaling to 52× at 256 cores. Random peaks at 144 cores, scaling to 91×, and drops to 49× at 256 cores. Hints scales to 186×, and LBHints performs best, with a 236× speedup at 256 cores.

Figure 3-19b yields further insights into these differences. The height of each bar in Figure 3-19b is the sum of cycles spent by all cores, normalized to the cycles of Random (lower is better). Each bar shows the breakdown of cycles spent executing tasks that are ultimately committed, eventually aborted, cycles stalled on a full queue, and spent in other overheads. Most cycles are spent running committed tasks, aborted tasks, or in queue stalls, and trends are widely different across schemes.

Committed cycles mainly depend on locality: in the absence of conflicts, the only difference is memory stalls. Random has the highest committed cycles (most stalls), while Hints and LBHints have the lowest, as gates are held in nearby private caches. Stealing has slightly higher committed cycles, as it often keeps tasks for nearby gates in the same tile.

Differences in aborted cycles are higher. In des, conflict frequency depends highly on how closely tasks from different tiles follow timestamp order. Random and LBHints keep tiles running tasks with close-by timestamps. However, conflicts in LBHints are local, and thus much faster, and LBHints serializes tasks that operate on the same gate. For these reasons, LBHints spends the fewest cycles on aborts. Hints is less balanced, so it incurs more conflicts than LBHints. Finally, in Stealing, tiles run tasks widely out of order, as stealing from the most loaded tile is not a good strategy to maintain order in des (as we will see, this is a good strategy in other cases). This causes both significant aborts and queue stalls in Stealing, as commit queues fill up. These effects hinder Stealing's scalability.

Overall, these results show that hints can yield significant gains by reducing both aborts and data movement.

3.7 Spatial Task Mapping with Hints

We now describe *spatial hints*, a general technique that leverages application-level knowledge to achieve high-quality task mappings. A hint is simply an abstract integer value, given at task creation time, that denotes the data likely to be accessed by a task. Hardware leverages hints to map tasks likely to access the same data to the same location.

Hints are conveyed as shown in the API described in Section 3.3.1. The hint can take one of three values:

- A 64-bit integer value that conveys the data likely to be accessed. The programmer is free to choose what this integer represents (e.g., addresses, object ids, etc.). The only guideline is that tasks likely to access the same data should have the same hint.
- `NOHINT`, used when the programmer does not know what data will be accessed.
- `SAMEHINT`, which assigns the parent's hint to the child task.

We employ unused bits in the `enqueue_task` instruction opcode to represent whether the new task is tagged with an integer hint, `NOHINT`, or `SAMEHINT`. If tagged with an integer hint, we pass that value through another register.

3.7.1 Hardware Mechanisms

Hardware leverages hints in two ways:

1. Spatial task mapping: When a core creates a new task, the local task unit uses the hint to determine its destination tile. The task unit hashes the 64-bit hint down to a tile ID (e.g., 6 bits for 64 tiles), then sends the task descriptor to the selected tile. `SAMEHINT` tasks are queued to the local task queue, and `NOHINT` tasks are sent to a random tile.

2. Serializing conflicting tasks: Since two tasks with the same hint are likely to conflict, we enhance the task dispatch logic to avoid running them concurrently. Specifically, tasks carry a 16-bit hash of their hint throughout their lifetime. By default, the task unit selects the earliest-timestamp idle task for execution. Instead, we check whether that candidate task's hint hash matches one of the already-running tasks. If there is a match and the already-running task has an earlier timestamp, the task unit skips the candidate and tries the idle task with the next lowest timestamp.

Using 16-bit hashed hints instead of full hints requires less storage and simplifies the dispatch logic. Their lower resolution introduces a negligible false-positive match probability ($6 \cdot 10^{-5}$ with four cores per tile).

Overheads: These techniques add small overheads:

- 6- and 16-bit hash functions at each tile to compute the tile ID and hashed hint.
- An extra 16 bits per task descriptor. Descriptors are sent through the network (so hints add some traffic) and stored in task queues. In our chosen configuration, each tile's task queue requires 512 extra bytes.
- Four 16-bit comparators used during task dispatch.

3.7.2 Adding Hints to Benchmarks

	Task Funcs	Hint patterns
bfs	1	Cache line of vertex
sssp	1	Cache line of vertex
astar	1	Cache line of vertex
color	3	Cache line of vertex
des	8	Logic gate ID
nocsim	10	Router ID
silo	16	(Table ID, primary key)
genome	10	Elem addr, map key, NO/SAMEHINT
kmeans	5	Cache line of point, cluster ID

Table 3.6: Benchmark information: number of task functions, and hint patterns used.

We add hints to nine of the benchmarks described in Table 3.4. Table 3.6 summarizes the strategies used to assign hints to each benchmark. We observe that a few common *patterns* arise naturally when adding hints to these applications. We explain each of these patterns through a representative application.

Cache-line address: Our graph analytics applications (`bfs`, `sssp`, `astar`, and `color`) are vertex-centric [132]: each task operates on one vertex and visits its neighbors. For example, Listing 3.1 shows the single task function of `sssp`. Given the distance to the source of vertex `v`, the task visits each neighbor `n`; if the projected distance to `n` is reduced, `n`'s distance is updated and a new task created for `n`. Tasks appear to execute in timestamp order, i.e. the projected distance to the source.

```
void ssspTask(Timestamp pathDist, Vertex* v) {
  if (pathDist == v->distance)
    for (Vertex* n : v->neighbors) {
      uint64_t projected = pathDist + length(v,n);
      if (projected < n->distance) {
        n->distance = projected;
        swarm::enqueue(ssspTask,
                projected /*Timestamp*/,
                cacheLine(n) /*Hint*/, n);
      }
    }
}
```

Listing 3.1: Hint-tagged `sssp` task.

Each task's hint is the cache-line address of the vertex it visits. Every task iterates over its vertex's neighbor list. This incurs two levels of indirection: one from the vertex to walk its neighbor list, and another from each neighbor to access and modify the neighbor's distance. Using the line address of the vertex lets us perform all the accesses to each neighbor list from a single tile, improving locality; however, each distance is accessed from different

tasks, so hints do not help with those accesses. We use cache-line addresses because several vertices reside on the same line, allowing us to exploit spatial locality.

`bfs`, `astar`, and `color` have similar structure, so we also use the visited vertex's line address as the hint. The limiting aspect of this strategy is that it fails to localize a large fraction of accesses (e.g., to `distance` in `sssp`), because each task accesses state from multiple vertices. This coarse-grain structure is natural for software implementations (e.g., sequential and parallel Galois `sssp` are written this way), but we will later see that fine-grained versions make hints much more effective.

Object IDs: In `des` and `nocsim` each task operates on one system component: a logic gate, or an NoC router component (e.g. its VC allocator), respectively. Similar to the graph algorithms, a task creates children tasks for its neighbors. In contrast to graph algorithms, each task only accesses state from its own component.

We tag simulator tasks with the gate ID and router ID, respectively. In `des`, using the gate ID is equivalent to using its line address, as each gate spans one line. Since each `nocsim` task operates on a router component, using component IDs or addresses as hints might seem appealing. However, components within the same router create tasks (events) for each other very often, and share state (e.g., pipeline registers) frequently. We find it is important to keep this communication local to a tile, which we achieve by using the coarser router IDs as hints.

Abstract unique IDs: In `silo`, each database transaction consists of tens of tasks. Each task reads or updates a tuple in a specific table. This tuple's address is not known at task creation time: the task must first traverse a tree to find it. Thus, unlike in prior benchmarks, hints cannot be concrete addresses. However, we know enough information to uniquely identify the tuple at task creation time: its table and primary key. Therefore, we compute the task's hint by concatenating these values. This way, tasks that access same tuple map to the same tile.

NOHINT and SAMEHINT: In `genome`, we do not know the data that one of its transactions, `T`, will access when the transaction is created. However, `T` spawns other transactions that access the same data as `T`. Therefore, we enqueue `T` with `NOHINT`, and its children with `SAMEHINT` to exploit parent-child locality.

Multiple patterns: Several benchmarks have different tasks that require different strategies. For instance, `kmeans` has two types of tasks: `findCluster` operates on a single point, determining its closest cluster centroid and updating the point's membership; and `updateCluster` updates the coordinates of the new centroid. `findCluster` uses the point's cache line as a hint, while `updateCluster` uses the centroid's ID. `genome` also uses a variety of patterns, as shown in Table 3.6.

In summary, a task can be tagged with a spatial hint when some of the data it accesses can be identified (directly or abstractly) at task creation time. In all applications, integer hints are either addresses or IDs. Often, we can use either; we use whichever is easier to compute (e.g., if we already have a pointer to the object, we use addresses; if have its ID and would e.g., need to index into an array to find its address, we use IDs). It may be helpful to assign a *coarse* hint, i.e., one that covers more data than is accessed by the specific task, either to exploit spatial locality when tasks share the same cache line (e.g. `sssp`, `kmeans`), or to group tasks with frequent communication (e.g. `nocsim`).

3.8 Evaluation of Spatial Hints

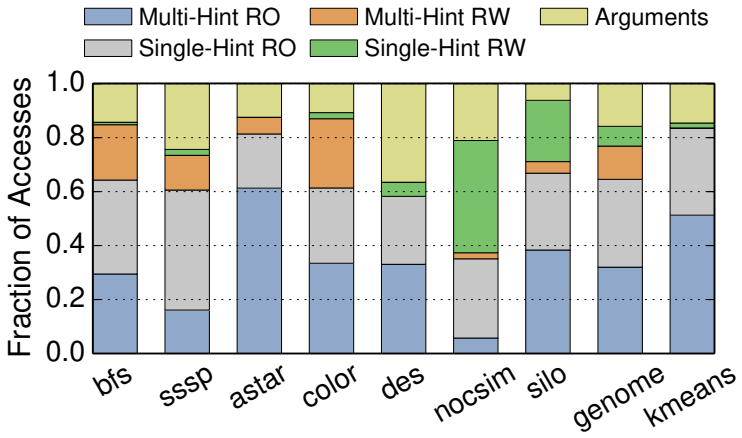


Figure 3-20: Classification of memory accesses.

3.8.1 Effectiveness of Hints

We first perform an architecture-independent analysis to evaluate the effectiveness of hints. We profile all the memory accesses made by committing tasks, and use this to classify each memory location in two dimensions: *read-only* vs. *read-write*, and *single-hint* vs. *multi-hint*. We classify data as read-only if, during its lifetime (from allocation to deallocation time), it is read at least 1000 times per write (this includes data that is initialized once, then read widely); we classify data as single-hint if more than 90% of accesses come from tasks of a single hint. We select fixed thresholds for simplicity, but results are mostly insensitive to their specific values.

Figure 3-20 classifies data accesses according to these categories. Each bar shows the breakdown of accesses for one application. We classify accesses in five types: those made to arguments,[1] and those made to non-argument data of each of the four possible types (multi-/single-hint, read-only/read-write).

[1]Swarm passes up to three 64-bit arguments per task through registers, and additional arguments through memory; this analysis considers both types of arguments equally.

Figure 3-20 reveals two interesting trends. First, on all applications, a significant fraction of read-only data is single-hint. Therefore, we expect hints to improve cache reuse by mapping tasks that use the same data to the same tile. All applications except `nocsim` also have a significant amount of multi-hint read-only accesses; often, these are accesses to a small amount of global data, which caches well. Second, hint effectiveness is more mixed for read-write data: in `des`, `nocsim`, `silo`, and `kmeans`, most read-write data is single-hint, while multi-hint read-write data dominates in `bfs`, `sssp`, `astar`, `color`, and `genome`. Read-write data is more critical, as mapping tasks that write the same data to the same tile not only improves locality, but also reduces conflicts.

In summary, hints effectively localize a significant fraction of accesses to read-only data, and, in 4 out of 9 applications, most accesses to read-write data (fine-grained versions in Section 3.9 will improve this to 8 out of 9). We now evaluate the impact of these results on performance.

3.8.2 Comparison of Schedulers

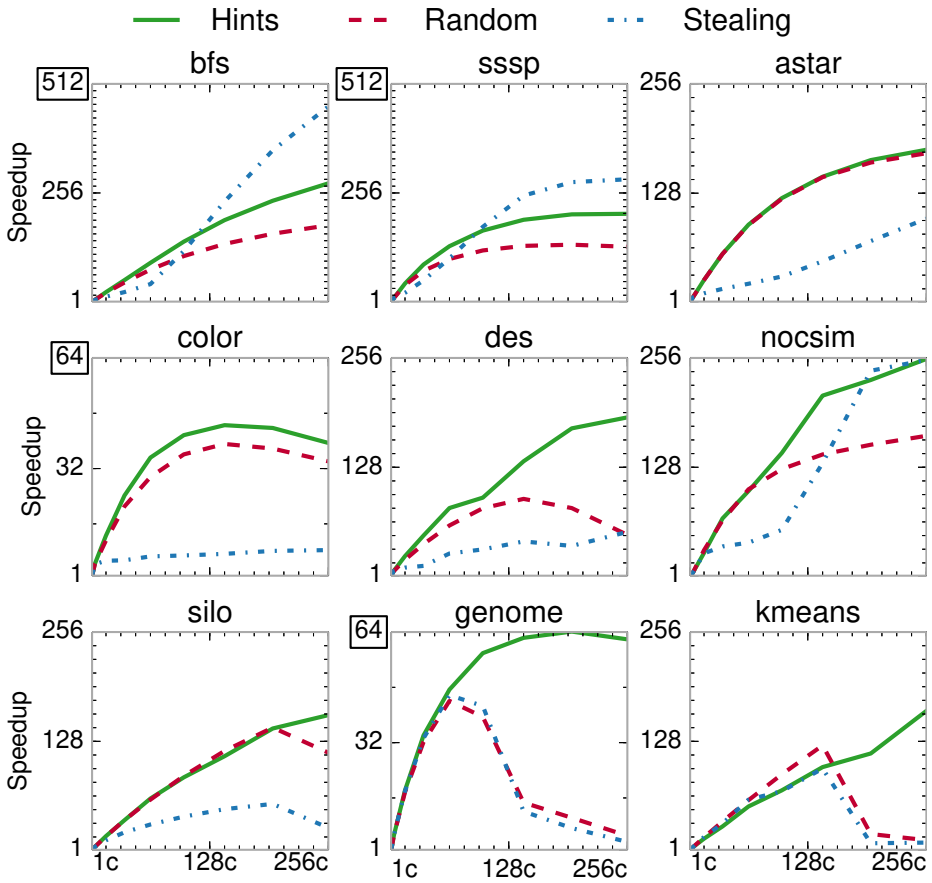
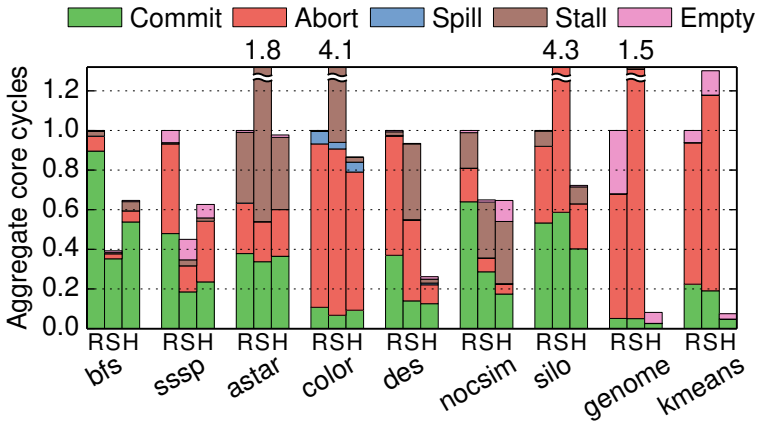


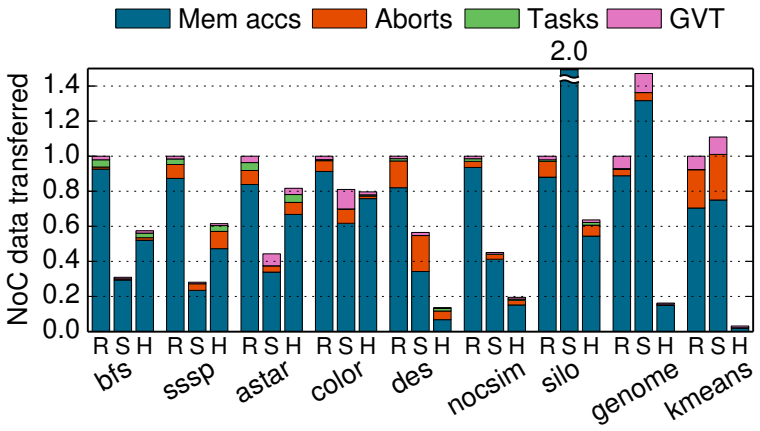
Figure 3-21: Speedup of different schedulers from 1 to 256 cores, relative to a 1-core system. We simulate systems with $K \times K$ tiles for $K \leq 8$.

Figure 3-21 compares the scalability of the Random, Stealing, and Hints schedulers on 1–256-core systems, similar to Figure 3-19a.

Overall, at 256 cores, Hints performs at least as well as Random (`astar`) or outperforms it by 16% (`color`) to 13× (`kmeans`). At 256 cores, Hints scales from 39.4× (`color`) to 279× (`bfs`). Hints outperforms Random across all core counts except on `kmeans` at 16–160 cores, where Hints is hampered by imbalance (hint-based load balancing will address this). While Hints and Random follow similar trends, Stealing's performance is spotty. On the one hand, Stealing is the best scheduler in `bfs` and `sssp`, outperforming Hints by up to 65%. On the other hand, Stealing is the worst scheduler in most other ordered benchmarks, and tracks Random on unordered ones.



(a) Breakdown of total core cycles



(b) Breakdown of total NoC data transferred

Figure 3-22: Breakdown of (a) core cycles and (b) NoC data transferred at 256 cores, under Random, Stealing, and Hints schedulers. Each bar is normalized to Random's.

Figure 3-22 gives more insight into these results by showing core cycle and network traffic breakdowns at 256 cores. Each bar of Figure 3-22a shows the breakdown of cycles spent *(i)* running tasks that are ultimately committed, *(ii)* running tasks that are later

aborted, *(iii)* spilling tasks from the hardware task queues, *(iv)* stalled on a full task or commit queue, or *(v)* stalled due to lack of tasks. Each bar of Figure 3-22b shows the breakdown of data transferred over the NoC (in total flits injected), including *(i)* memory accesses (between L2s and LLC, or LLC and main memory), *(ii)* abort traffic (including child abort messages and rollback memory accesses), *(iii)* tasks enqueued to remote tiles, and *(iv)* GVT updates (for commits). In both cases, results are relative to Random's. We first compare Hints and Random, then discuss Stealing.

Hints vs. Random: Beyond outperforming Random, Hints also improves efficiency, substantially reducing cycles wasted to aborted tasks and network traffic. Performance and efficiency gains are highly dependent on the fraction of accesses to single-hint data (Section 3.8.1).

In graph analytics benchmarks, Hints achieves negligible (`astar`) to moderate improvements (`bfs`, `sssp`, `color`). `bfs`, `sssp`, and `color` have a substantial number of single-hint read-only accesses. These accesses cache well, reducing memory stalls. This results in a lower number of committed-task cycles and lower network traffic. However, cycles wasted on aborted tasks barely change, because nearly all accesses to contentious read-write data are multi-hint. In short, improvements in these benchmarks stem from locality of single-hint read-only accesses; since these are infrequent in `astar`, Hints barely improves its performance.

In `des`, `nocsim`, and `silo`, Hints significantly outperforms Random, from 1.4× (`silo`) to 3.8× (`des`). In these benchmarks, many read-only *and* most read-write accesses are to single-hint data. As in graph analytics benchmarks, Hints reduces committed cycles and network traffic. Moreover, aborted cycles and network traffic drop dramatically, by up to 6× and 7× (`des`), respectively. With Hints, these benchmarks are moderately limited by load imbalance, which manifests as stalls in `nocsim` and aborts caused by running too far-ahead tasks in `des` and `silo`.

Hints has the largest impact on the two unordered benchmarks, `genome` and `kmeans`. It outperforms Random by up to 13× and reduces network traffic by up to 32× (`kmeans`). For `kmeans`, these gains arise because Hints localizes and serializes all single-hint read-write accesses to the small amount of highly-contended data (the K cluster centroids). However, co-locating the many accessor tasks of one centroid to one tile causes imbalance. This manifests in two ways: *(i)* Random outperforms Hints from 16–160 cores in Figure 3-21, and *(ii)* empty stalls are the remaining overhead at 256 cores. Hint-based load balancing addresses this problem (Section 3.10). In contrast to `kmeans`, `genome` has both single- and multi-hint read-write data, but Hints virtually eliminates aborts. Accesses to multi-hint read-write data rarely contend, while accesses to single-hint read-write data are far more contentious. Beyond 64 cores, both schedulers approach the limit of concurrency, dominated by an application phase with low parallelism; this phase manifests as empty cycles in Figure 3-22a.

Stealing: Stealing shows disparate performance across benchmarks, despite careful tuning and idealizations (Section 3.6). Stealing suffers from two main pathologies. First, Stealing often fails to keep tiles running tasks of roughly similar timestamps, which hurts several ordered benchmarks. Second, when few tasks are available, Stealing moves tasks across tiles too aggressively, which hurts the unordered benchmarks.

Interestingly, although they are ordered, `bfs` and `sssp` perform best under Stealing. Because most visited vertices expand the fringe of vertices to visit, Stealing manages to keep tiles balanced with relatively few steals, and keeps most tasks for neighboring nodes in the same tile. Because each task accesses a vertex and its neighbors (Listing 3.1), Stealing enjoys good locality, achieving the lowest committed cycles and network traffic. `bfs` and `sssp` tolerate Stealing's looser cross-tile order well, so Stealing outperforms the other schedulers.

Stealing performs poorly in other ordered benchmarks. This happens because stealing the earliest task from the most loaded tile is insufficient to keep all tiles running tasks with close-by timestamps. Instead, some tiles run tasks that are too far ahead in program order. In `astar` and `color`, this causes a large increase in commit queue stalls, which dominate execution. In `des` and `silo`, this causes both commit queue stalls and aborts, as tasks that run too early mispeculate frequently. `nocsim` also suffers from commit queue stalls and aborts, but to a smaller degree, so Stealing outperforms Random but underperforms Hints at 256 cores.

By contrast, `genome` and `kmeans` are unordered, so they do not suffer from Stealing's loose cross-tile order. Stealing tracks Random's performance up to 64 cores. However, these applications have few tasks per tile at large core counts, and Stealing underperforms Random because it rebalances tasks too aggressively. In particular, it sometimes steals tasks that have already run, but have aborted. Rerunning these aborted tasks at the same tile, as Random does, incurs fewer misses, as the tasks have already built up locality at the tile.

3.9 Improving Locality and Parallelism with Fine-Grained Tasks

We now analyze the relationship between task granularity and hint effectiveness. We show that programs can often be restructured to use finer-grained tasks, which make hints more effective.

For example, consider the coarse-grained implementation of `sssp` in Listing 3.1. This implementation causes vertex distances to be read and written from multiple tasks, which renders hints ineffective for read-write data. Instead, Listing 3.2 shows an equivalent version of `sssp` where each task operates on the data (distance and neighbor list) of a single node.

Instead of setting the distances of all its neighbors, this task launches one child task per neighbor. Each task accesses its own distance. This transformation generates substantially

```
void ssspTaskFG(Timestamp pathDist, Vertex* v) {
 if (v->distance == UNSET) {
  v->distance = pathDist;
  for (Vertex* n : v->neighbors)
   swarm::enqueue(ssspTaskFG,
           pathDist + length(v,n) /*Timestamp*/,
           cacheLine(n) /*Hint*/, n);
 }
}
```

Listing 3.2: Fine-grain `sssp` implementation.

more tasks, as each vertex is often visited multiple times. In a serial or parallel version with software scheduling, the coarse-grained approach is more efficient, as a memory access is cheaper than creating additional tasks. But *in large multicores with hardware scheduling, this tradeoff reverses*: sending a task across the chip is cheaper than incurring global conflicts and serialization.

We have also adapted three other benchmarks with significant multi-hint read-write accesses. `bfs` and `astar` follow a similar approach to `sssp`. In `color`, each task operates on a vertex, reads from all neighboring vertices, and updates its own; our fine-grained version splits this operation into four types of tasks, each of which reads or writes at most one vertex.

We do not consider finer-grain versions of `des`, `nocsim`, `silo`, or `kmeans` because they already have negligible multi-hint read-write accesses, and it is not clear how to make their tasks smaller. We believe a finer-grain `genome` would be beneficial, but this would require turning it into an ordered program to be able to break transactions into smaller tasks while retaining atomicity [114].

Tradeoffs: In general, fine-grained tasks yield two benefits: *(i)* improved parallelism, and, *(ii)* with hints, improved locality and reduced conflicts. However, fine-grained tasks also introduce two sources of overhead: *(i)* additional work (e.g., when a coarse-grained task is broken into multiple tasks, several fine-grained tasks may need to read or compute the same data), and *(ii)* more pressure on the scheduler.

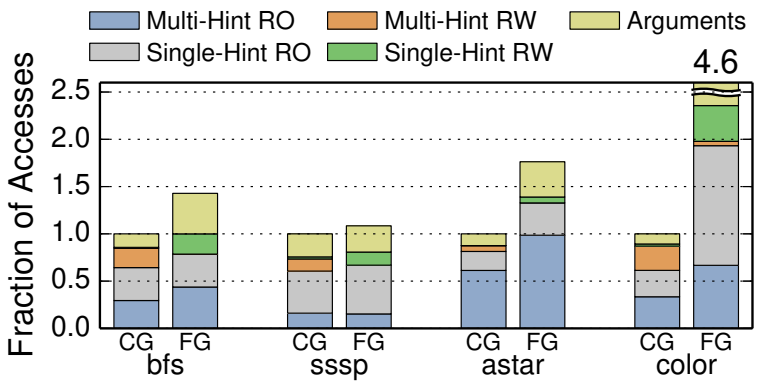


Figure 3-23: Classification of memory accesses for coarse-grained (CG) and fine-grained (FG) versions.

Effectiveness of Hints: Figure 3-23 compares the memory accesses of coarse-grained (CG) and fine-grained (FG) versions. Figure 3-23 is similar to Figure 3-20, but bars are normalized to the CG version, so the height of each FG bar denotes how many more accesses it makes. Figure 3-23 shows that FG versions make hints much more effective: virtually all accesses to read-write data become single-hint, and more read-only accesses become single-hint. Nevertheless, this comes at the expense of extra accesses (and work): from 8% more accesses in `sssp`, to 4.6× more in `color` (2.6× discounting arguments).

3.9.1 Evaluation

Figure 3-24 compares the scalability of CG and FG versions under the three schedulers. Speedups are relative to the CG versions at one core. Figure 3-25 shows cycle and network traffic breakdowns, with results normalized to CG under Random (as in Figure 3-22). Overall, FG versions improve Hints uniformly, while they have mixed results with Random and Stealing.

In `bfs` and `sssp`, FG versions improve scalability and reduce data movement, compensating for their moderate increase in work. Figure 3-25a shows that Hints improve locality (fewer committed cycles) and reduce aborts. As a result, FG versions under Hints incur much lower traffic (Figure 3-25b), up to 4.8× lower than CG under Hints and 7.7× lower than CG under Random in `sssp`.

`astar`'s FG version does not outperform CG: though it reduces aborts, the smaller tasks stress commit queues more, increasing stalls (Figure 3-25a). Nonetheless, FG improves efficiency and reduces traffic by 61% over CG under Hints (Figure 3-25b).

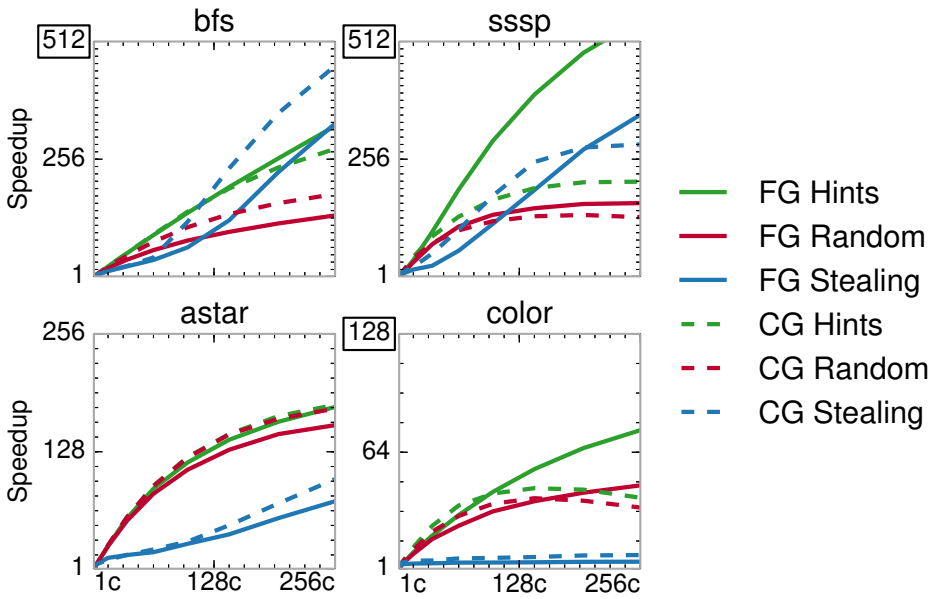


Figure 3-24: Speedup of fine-grained (FG) and coarse-grained (CG) versions, relative to CG versions on a 1-core system.

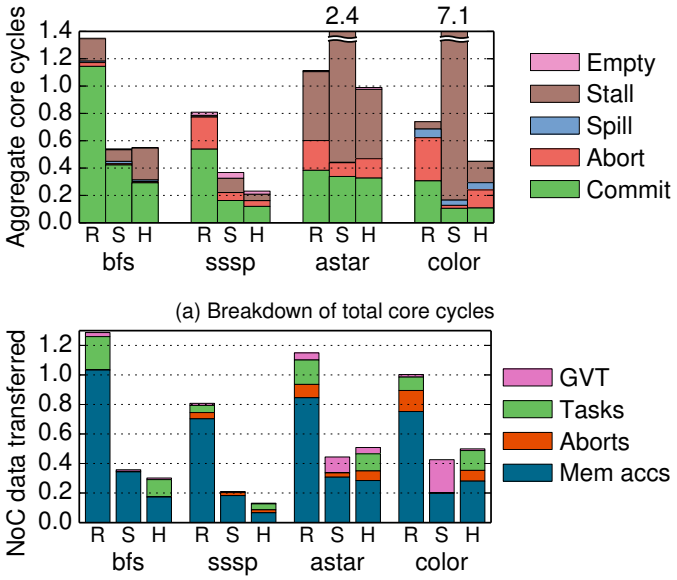


Figure 3-25: Breakdown of (a) core cycles and (b) NoC data transferred in fine-grained versions at 256 cores, under Random, Stealing, and Hints. Each bar is normalized to the coarse-grained version under Random (as in Figure 3-22).

`color`'s FG version performs significantly more work than CG, which is faster below 64 cores. Beyond 64 cores, however, FG reduces aborts dramatically (Figure 3-25a), outperforming CG under Hints by 93%.

Finally, comparing the relative contributions of tasks sent in Figure 3-25b vs. Figure 3-22b shows that, although the amount of task data transferred across tiles becomes significant, the reduction of memory access traffic more than offsets this scheduling overhead.

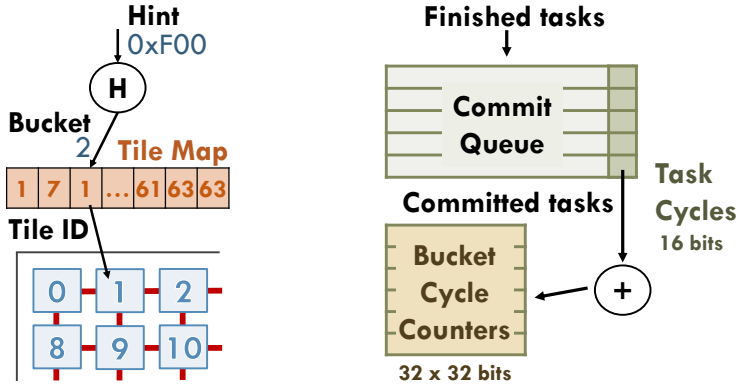
In summary, fine-grained versions substantially improve the performance and efficiency of Hints. This is not always the case for Random or Stealing, as they do not exploit the locality benefits of fine-grained tasks.

3.10 Data-Centric Load Balancing

While hint-based task mapping improves locality and reduces conflicts, it may cause load imbalance. For example, in `nocsim`, routers in the middle of the simulated mesh handle more traffic than edge routers, so more tasks operate on them, and their tiles become overloaded. We address this problem by dynamically remapping hints across tiles to equalize their load.

We have designed a new load balancer because non-speculative ones work poorly. For example, applying stealing to hint-based task mapping hurts performance. The key reason is that *load is hard to measure*: non-speculative schemes use queued tasks as a proxy for

load, but with speculation, underloaded tiles often do not run out of tasks—rather, they run too far ahead and suffer more frequent aborts or full-queue stalls.



(a) Configurable tile map (b) Profiling committed cycles

Figure 3-26: Hardware modifications of hint-based load balancer.

Instead, we have found that the number of committed cycles is a better proxy for load. Therefore, our load balancer remaps hints across tiles to *balance their committed cycles per unit time*. Our design has three components:

1. Configurable hint-to-tile mapping with buckets: Instead of hashing a hint to produce a tile ID directly, we introduce a reconfigurable level of indirection. As shown in Figure 3-26(a), when a new task is created, the task unit hashes its hint to produce a *bucket*, which it uses to index into a *tile map* and obtain the destination tile's ID, to which it sends the task.

The tile map is a table that stores one tile ID for every bucket. To achieve fine-enough granularity, the number of buckets should be larger than the number of tiles. We find 16 buckets/tile works well, so at 256 cores (64 tiles) we use a 1024-bucket tile map. Each tile needs a fixed 10-bit hint-to-bucket hash function and a 1024×6-bit tile map (768 bytes).

We periodically reconfigure the tile map to balance load. The mapping is static between reconfigurations, allowing tasks to build locality at a particular tile.

2. Profiling committed cycles per bucket: Accurate reconfigurations require profiling the distribution of committed cycles across buckets. Each tile profiles cycles locally, using three modifications shown in Figure 3-26(b). First, like the hashed hint (Section 3.7.1), tasks carry their bucket value throughout their lifetime. Second, when a task finishes execution, the task unit records the number of cycles it took to run. Third, if the task commits, the tile adds its cycles to the appropriate entry of the per-bucket committed cycle counters.

A naive design would require each tile to have as many committed cycle counters as buckets (e.g., 1024 at 256 cores). However, each tile only executes tasks from the buckets that map to it; this number of mapped buckets is 16 per tile on average. We implement the committed cycle counters as a tagged structure with enough counters to sample 2×

this average (i.e., 32 counters in our implementation). Overall, profiling hardware takes ∼600 bytes per tile.

3. Reconfigurations: Initially, the tile map divides buckets uniformly among tiles. Periodically (every 500 Kcycles in our implementation), a core reads the per-bucket committed cycle counters from all tiles and uses them to update the tile map, which it sends to all tiles.

The reconfiguration algorithm is straightforward. It computes the total committed cycles per tile, and sorts tiles from least to most loaded. It then greedily donates buckets from overloaded to underloaded tiles. To avoid oscillations, the load balancer does not seek to completely equalize load at once. Rather, an underloaded tile can only reduce its deficit (difference from the average load) by a fraction f (80% in our implementation). Similarly, an overloaded tile can only reduce its surplus by a fraction f. Reconfigurations are infrequent, and the software handler completes them in ∼50 Kcycles (0.04% of core cycles at 256 cores).

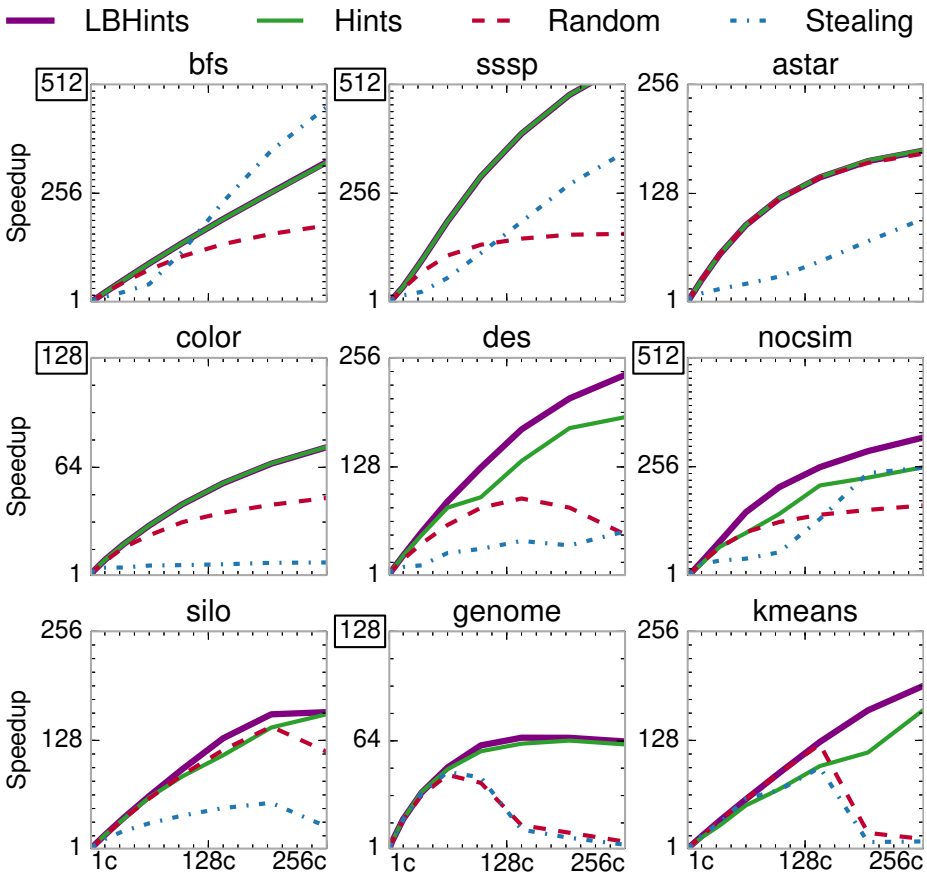


Figure 3-27: Speedup of Random, Stealing, Hints, and LBHints schedulers from 1 to 256 cores, relative to a 1-core system. For applications with coarse- and fine-grained versions, we report the best-performing version for each scheme.

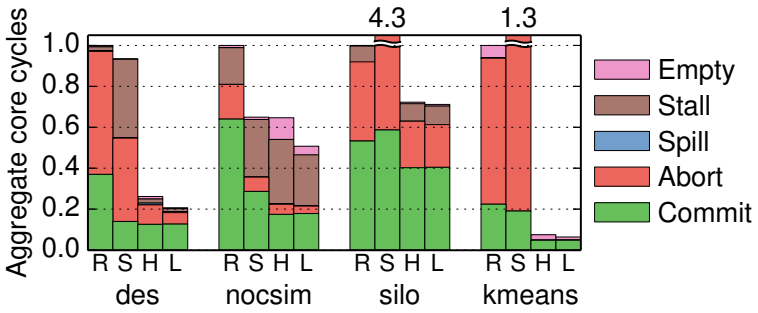


Figure 3-28: Breakdown of total core cycles at 256 cores under Random, Stealing, Hints, and LBHints.

3.10.1 Evaluation

Figure 3-27 reports the scalability of applications with our hint-based load balancer, denoted LBHints. LBHints improves performance on four applications, and neither helps nor hurts performance on the other five.

In `des`, LBHints outperforms Hints by 27%, scaling to 236×. As described in Section 3.6, in `des` load imbalance causes aborts as some tiles run too far ahead; LBHints's gains come from reducing aborts, as shown in Figure 3-28. In `nocsim`, LBHints outperforms Hints by 27%, scaling to 325×, and in `kmeans`, LBHints outperforms Hints by 17%, scaling to 192×. These gains come from reducing empty stalls, commit queue stalls, and aborts. Finally, LBHints improves `silo` by a modest 1.5% at 256 cores, and by 18% at 144 cores. In all cases, LBHints does not sacrifice locality (same committed cycles as Hints) and yields smooth speedups.

Finally, we also evaluated using other signals as a proxy for load in the hint-based load balancer. Using the number of idle tasks in each tile to estimate load performs significantly worse than LBHints. At 256 cores, this variant improves performance over Hints by 12% on `nocsim` and 2% on `silo`, and degrades performance by 9% on `des` and 1.2% on `kmeans`. This happens because balancing the number of idle tasks does not always balance the amount of useful work across tiles.

3.10.2 Putting It All Together

Together, the techniques we have presented make speculative parallelism practical at large core counts. Across our nine applications, Random achieves 58× gmean speedup at 256 cores; Hints achieves 146×; with the fine-grained versions from Section 3.9 instead of their coarse-grained counterparts, Hints scales to 179× (while Random scales to 62× only); and LBHints scales to 193× gmean.[2]

[2]The corresponding harmonic-mean speedups are 25× for Random, 117× for Hints, 146× for Hints with fine-grained versions, and 154× for LBHints.

3.11 Additional Related Work

3.11.1 Limits of Parallelism

Prior work has studied the limits of instruction-level parallelism under several idealizations, including a large or infinite instruction window, perfect branch prediction and memory disambiguation, and simple program transformations to remove unnecessary data dependences [17, 39, 71, 74, 83, 122, 145, 158, 211]. Similar to our limit study, these analyses find that parallelism is often plentiful (>1000×), but very large instruction windows are needed to exploit it (>100K instructions [122, 158]). Our oracle tool focuses on task-level parallelism, so it misses intra-task parallelism, which is necessarily limited with short tasks. Instead, we focus on removing superfluous dependences in scheduling data structures, uncovering large amounts of parallelism for ordered applications.

3.11.2 Thread-Level Speculation

Several TLS schemes expose timestamps to software for different purposes, such as letting the compiler schedule loop iterations in Stampede [195], speculating across barriers in TCC [95], and supporting out-of-order spawn of speculative function calls in Renau et al. [171]. These schemes work well for their intended purposes, but cannot queue or buffer tasks with arbitrary timestamps—they can only spawn new work if there is a free hardware context. Software scheduling would be required to sidestep this limitation, which, as we have seen, would introduce false data dependences and limit parallelism.

3.11.3 Scalable Software Priority Queues

Prior work has developed shared-memory priority queues that scale with the number of cores [7, 214], but they do so by relaxing priority order. This restricts them to benchmarks that admit order violations, and loss of order means threads often execute useless work far from the critical path [100, 101]. Nikas et al. [147] use hardware transactional memory to partially parallelize priority queue operations, accelerating `sssp` by 1.8× on 14 cores. Instead, we dispense with shared-memory priority queues: Swarm uses distributed and load-balanced priority queues, and uses speculation to maintain order.

3.11.4 Scheduling in Speculative Parallelism

Speculative execution models have seen relatively little attention with regards to optimizing locality.

Ordered parallelism: Thread-Level Speculation (TLS) schemes dispatch tasks to threads as they become available, without concern for locality [84, 94, 170, 171, 191, 196]. TLS

schemes targeted systems with few cores, but cache misses hinder TLS scalability even at small scales [82].

Unordered parallelism: TM programs are commonly structured as threads that execute a fixed sequence of transactions [42, 95, 137]. Prior work has observed that it is often beneficial to structure code as a collection of transactional tasks, and schedule them across threads using a variety of hardware and software techniques [14,15,23,24,66,68,177,221]. Prior transactional schedulers focus on *limiting concurrency*, not spatial task mapping. These schemes are either reactive or predictive. ATS [221], CAR-STM [67], and Steal-on-Abort [14] serialize aborted transactions after the transaction they conflicted with, avoiding repeated conflicts. PTS [23], BFGTS [24], Shrink [68], and HARP [15] instead predict conflicts by observing the read- and write-sets of prior transactions, and serialize transactions that are predicted to conflict. Unlike predictive schemes, we avoid conflicts by leveraging program hints. Hints reduce conflicts more effectively than prior predictive schedulers, and require much simpler hardware. Moreover, unlike prior transactional schedulers, our approach does not rely on centralized scheduling structures or frequent broadcasts, so it scales to hundreds of cores.

A limitation of our approach vs. predictive transaction schedulers is that programmers must specify hints. We have shown that it is easy to provide accurate hints. It may be possible to automate this process, e.g. through static analysis or profile-guided optimization; we defer this to future work.

Data partitioning: Kulkarni et al. [119] propose a software speculative runtime that exploits partitioning to improve locality. Data structures are statically divided into a few coarse partitions, and partitions are assigned to cores. The runtime maps tasks that operate on a particular partition to its assigned core, and reduces overheads by synchronizing at partition granularity. Schism [59] applies a similar approach to transactional databases. These techniques work well only when data structures can be easily divided into partitions that are both balanced and capture most parent-child task relations, so that most enqueues do not leave the partition. Many algorithms do not meet these conditions. While we show that simple hint assignments that do not rely on careful static partitioning work well, more sophisticated mappings may help some applications. For example, in `des`, mapping adjacent gates to nearby tiles may reduce communication, at the expense of complicating load balancing. We leave this exploration to future work.

Distributed transactional memory: Prior work has proposed STMs for distributed systems [33, 106, 176]. Some of these schemes, like ClusterSTM [33], allow migrating a transaction across machines instead of fetching remotely-accessed data. However, their interface is more onerous than hints: in ClusterSTM, programmers must know exactly how data is laid out across machines, and must manually migrate transactions across specific processors. Moreover, these techniques are dominated by the high cost of remote accesses and migrations [33], so load balancing is not an issue.

3.11.5 Scheduling in Non-Speculative Parallelism

In contrast to speculative models, prior work for non-speculative parallelism has developed many techniques to improve locality, often tailored to specific program traits [2, 4, 29, 48, 107, 141, 187, 218, 220]. Work-stealing [2, 29] is the most widely used technique. Work-stealing attempts to keep parent and child tasks together, which is near-optimal for divide-and-conquer algorithms, and as we have seen, minimizes data movement in some benchmarks (e.g., `bfs` and `sssp` in Section 3.8). Due to its low overheads, work-stealing is the foundation of most parallel runtimes [70, 118, 168], which extend it to improve locality by stealing from nearby cores or limiting the footprint of tasks [2, 91, 179, 220], or to implement priorities [125, 143, 179]. Prior work within the Galois project [100, 125, 155] has found that irregular programs (including software-parallel versions of several of our benchmarks) are highly sensitive to scheduling overheads and policies, and has proposed techniques to synthesize adequate schedulers [143, 160]. Likewise, we find that work-stealing is sensitive to the specific policies it uses.

In contrast to these schemes, we have shown that a simple hardware task scheduling policy can provide robust, high performance across a wide range of benchmarks. Hints enable high-quality spatial mappings and produce a balanced work distribution. Hardware task scheduling makes hints practical. Whereas a software scheduler would spend hundreds of cycles per remote enqueue on memory stalls and synchronization, a hardware scheduler can send short tasks asynchronously, incurring very low overheads on tasks as small as few tens of instructions. Prior work has also investigated hardware-accelerated scheduling, but has done so in the context of work-stealing [120, 181] and domain-specific schedulers [90, 215].

3.12 Summary

We have presented Swarm, a novel architecture that unlocks abundant but hard-to-exploit ordered parallelism. Swarm relies on a novel execution model based on timestamped tasks that decouples task creation and execution order, and a microarchitecture that performs speculative, out-of-order task execution and implements a large speculation window efficiently. Swarm achieves order-of-magnitude speedups on programs with ordered parallelism, which are key in emerging domains such as graph analytics, data mining, and in-memory databases.

Swarm defies conventional wisdom in two ways. First, conventional wisdom says that order constraints fundamentally limit parallelism. However, we have shown that it is possible to maintain a large speculation window efficiently, so that only true data dependencies limit parallelism. Second, now that energy efficiency is the key challenge facing computer systems, conventional wisdom says we should shun speculative parallelization, which wastes energy on mispeculations. However, we have shown that speculation unlocks plentiful ordered parallelism, which can be traded for efficiency in many ways, e.g., through simpler cores or slower clocks.

Finally, Swarm shows that hardware-software co-design is a promising strategy to unlock challenging types of parallelism and simplify parallel programming.

4

Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism

This work was conducted in collaboration with Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor Ying, Joel Emer and Daniel Sanchez from MIT. The Fractal execution model was developed collaboratively. This thesis contributes the hardware implementation of Fractal, and the mechanisms and implementation of zooming.

SYSTEMS that support speculative parallelization, such as hardware transactional memory (HTM) or thread-level speculation (TLS), have two major benefits over non-speculative systems: they uncover abundant parallelism in many challenging applications [95, 114] and simplify parallel programming [173]. But these systems suffer from limited support for *nested speculative parallelism*, i.e., the ability to invoke a speculative parallel algorithm within another speculative parallel algorithm. This causes three problems. First, it sacrifices substantial parallelism and limits the algorithms supported by these systems. Second, it disallows composing parallel algorithms, making it hard to write modular parallel programs. Third, it biases programmers to write coarse-grained speculative tasks, which are more expensive to support in hardware.

For example, consider the problem of parallelizing a transactional database. A natural approach is to use HTM and to make each database transaction a memory transaction. Each transaction executes on a thread, and the HTM system guarantees atomicity among concurrent transactions, detecting conflicting loads and stores on the fly, and aborting transactions to avoid serializability violations.

Unfortunately, this HTM approach faces significant challenges. First, each transaction must run on a single thread, but database transactions often consist of many queries or updates that could run in parallel. The HTM approach thus sacrifices this intra-transaction, fine-grained parallelism. Second, long transactions often have large read and write sets, which make conflicts and aborts more likely. These aborts often waste many operations

that were not affected by the conflict. Third, supporting large read/write sets in hardware is costly. Hardware can track small read/write sets cheaply, e.g., using private caches [95,191] or small Bloom filters [42, 219]. But these tracking structures have limited capacity and force transactions that overflow them to serialize, even when they have no conflicts [30, 95, 219]. Beyond these problems, HTM's unordered execution semantics are insufficient for programs with ordered parallelism, where speculative tasks must appear to execute in a program-specified order [114].

The Swarm architecture (Chapter 3) can address some of these problems. By exposing timestamps to programs, Swarm can parallelize more algorithms than prior ordered speculation techniques, like TLS; Swarm also supports unordered, HTM-style execution. As a result, Swarm often uncovers abundant fine-grained parallelism. But Swarm's software-visible timestamps can only convey very limited forms of nested parallelism, and they cause two key issues in this regard (Section 4.1). Timestamps make nested algorithms *hard to compose*, as algorithms at different nesting levels must agree on a common meaning for the timestamp. Timestamps also *over-serialize* nested algorithms, as they impose more order constraints than needed.

For instance, in the example above, Swarm can be used to break each database transaction into many small, ordered tasks. This exploits intra-transaction parallelism, and, at 256 cores, it is 21× faster than running operations within each transaction serially (Section 4.1.2). However, to maintain atomicity among database transactions, the programmer must needlessly order database transactions and must carefully assign timestamps to tasks within each transaction.

These problems are far from specific to database transactions. In general, large programs have speculative parallelism at multiple levels and often intermix ordered and unordered algorithms. Speculative architectures should support composition of ordered and unordered algorithms to convey all this nested parallelism without undue serialization.

We present two main contributions that achieve these goals. Our first contribution is *Fractal*, a new execution model for nested speculative parallelism. Fractal programs consist of tasks located in a hierarchy of nested *domains*. Within each domain, tasks can be ordered or unordered. Any task can create a new subdomain and enqueue new tasks in that subdomain. All tasks in a domain appear to execute atomically with respect to tasks outside the domain.

Fractal allows seamless composition of ordered and unordered nested parallelism. In the above example, each database transaction starts as a single task that runs in an unordered, root domain. Each of these unordered tasks creates an ordered subdomain in which it enqueues tasks for the different operations within the transaction. In the event of a conflict between tasks in two different transactions, Fractal selectively aborts conflicting tasks, rather than aborting all tasks in any one transaction. In fact, other tasks from the two transactions may continue to execute in parallel.

Our second contribution is a simple implementation of Fractal that builds on Swarm and supports arbitrary nesting levels cheaply (Section 4.3). Our implementation focuses on extracting parallelism at the finest (deepest) levels first. This is in stark contrast with current

HTMs. Most HTMs only support serial execution of nested transactions, forgoing intra-transaction parallelism. A few HTMs support parallel nested transactions [20,206], but they parallelize at the coarsest levels, suffer from subtle deadlock and livelock conditions, and impose large overheads because they merge the speculative state of nested transactions [19, 20]. The Fractal execution model lets our implementation avoid these problems. Beyond exploiting more parallelism, focusing on fine-grained tasks reduces the hardware costs of speculative execution.

We demonstrate Fractal's performance and programmability benefits through several case studies (Section 4.1) and a broad evaluation (Section 4.5). Fractal uncovers abundant fine-grained parallelism on large programs. For example, ports of the STAMP benchmark suite to Fractal outperform baseline HTM implementations by up to 88× at 256 cores. As a result, while several of the original STAMP benchmarks cannot reach even 10× scaling, Fractal makes all STAMP benchmarks scale well to 256 cores.

4.1 Motivation

We motivate Fractal through three case studies that highlight its key benefits: uncovering abundant parallelism, improving programmability, and avoiding over-serialization. Since Fractal subsumes prior speculative execution models (HTM, TLS, and Swarm), all case studies use the Fractal architecture (Section 4.3), and we compare applications written in Fractal vs. other execution models. This approach lets us focus on the effect of different Fractal features. Our implementation does not add overheads to programs that do not use Fractal's features.

4.1.1 Fractal Uncovers Abundant Parallelism

Consider the maxflow problem, which finds the maximum amount of flow that can be pushed from a source to a sink node in a network (a graph with directed edges labeled with capacities). Push-relabel is a fast and widely used maxflow algorithm [50], but it is hard to parallelize [22, 155]. 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.

To be efficient, push-relabel must use a heuristic that periodically recomputes node heights. Global relabeling [50] is a commonly used heuristic that updates many node heights by performing a breadth-first search on a subset of the graph. Global relabeling takes a significant fraction of the total work, typically 10–40% of instructions [13].

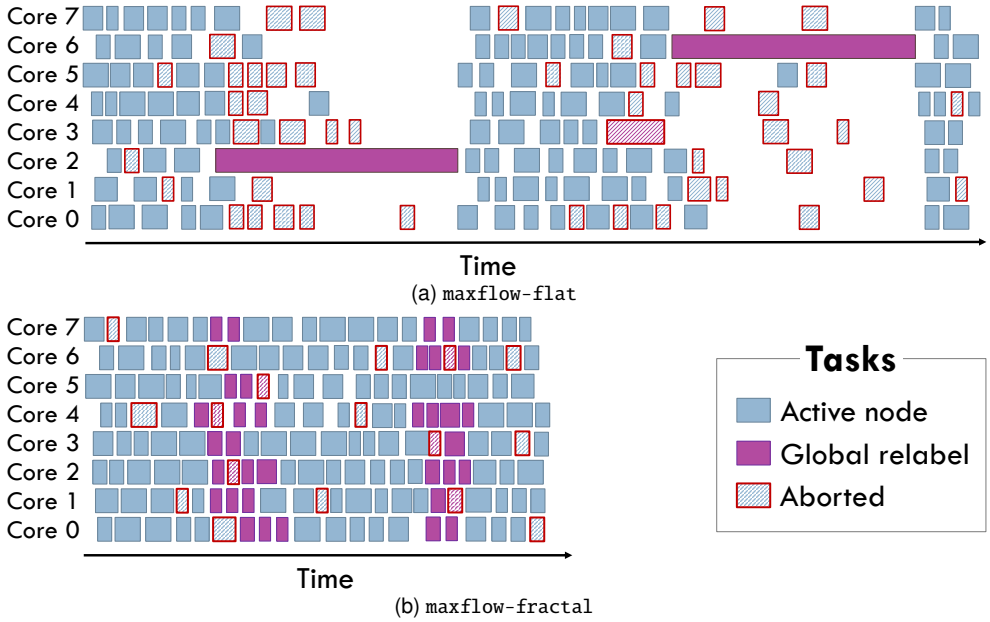


Figure 4-1: Execution timeline of (a) `maxflow-flat`, which consists of unordered tasks and does not exploit nested parallelism, and (b) `maxflow-fractal`, which exploits the nested ordered parallelism within global relabel.

Since push-relabel can process active nodes in an arbitrary order, it can be parallelized using transactional tasks of two types [155, 167]. An **active-node** task processes a single node, and may enqueue other tasks to process newly-activated nodes. A **global-relabel** task performs a global relabel operation. Each task must run atomically, since tasks access data from multiple neighbors and must observe a consistent state. We call this implementation `maxflow-flat`.

We simulate `maxflow-flat` on systems of up to 256 cores. (See Section 4.4 for methodology details.) At 256 cores, `maxflow-flat` scales to 4.9× only. Figure 4-1a illustrates the reason for this limited speedup: while **active-node** tasks are short, each **global-relabel** task is long, and queries and updates many nodes. When a **global-relabel** task runs, it conflicts with and serializes many **active-node** tasks.

Fortunately, each **global-relabel** task performs a breadth-first search, which has plentiful *ordered* speculative parallelism. Fractal lets us exploit this nested parallelism, running the breadth-first search in parallel while maintaining its atomicity with respect to other **active-node** tasks. To achieve this, we develop a `maxflow-fractal` implementation where each **global-relabel** task creates an ordered subdomain, in which it executes a parallel breadth-first search using fine-grained ordered tasks, as shown in Figure 4-2. A **global-relabel** task and its subdomain appear as a single atomic unit with respect to other tasks in the (unordered) root domain. Figure 4-1b illustrates how this improves parallelism and efficiency. As a result, Figure 4-3 shows that `maxflow-fractal` achieves a speedup of 322× at 256 cores (over `maxflow-flat` on one core).

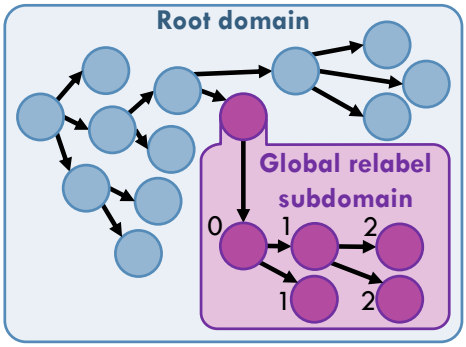


Figure 4-2: In `maxflow-fractal`, each **global-relabel** task creates an ordered subdomain.

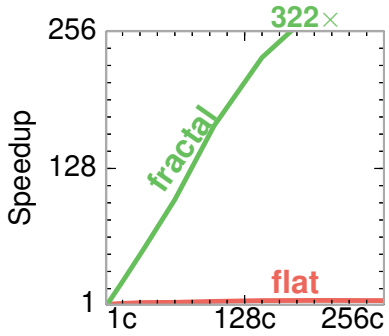


Figure 4-3: Speedup of different `maxflow` versions on 1–256 cores.

Fractal is the first architecture that effectively exploits maxflow's fine-grained nested parallelism: neither HTM, nor TLS, nor Swarm can support the combination of unordered and ordered parallelism maxflow has. Prior software-parallel push-relabel algorithms attempted to exploit this fine-grained parallelism [13, 155, 167], but the overheads of software speculation and scheduling negated the benefits of additional parallelism (in `maxflow-fractal`, each task is 373 cycles on average). We also evaluated two state-of-the-art software implementations: `prsn` [22] and Galois [155]. On 1–256 cores, they achieve maximum speedups of only 4.9× and 8.3× over `maxflow-flat` at one core, respectively.

4.1.2 Fractal Eases Parallel Programming

Beyond improving performance, Fractal's support for nested parallelism eases parallel programming because it enables *parallel composition*. Programmers can write multiple self-contained, modular parallel algorithms and compose them without sacrificing performance: when a parallel algorithm invokes another parallel algorithm, Fractal can exploit parallelism at both caller and callee.

In the previous case study, only Fractal was able to uncover nested parallelism. In some applications, prior architectures can also exploit the nested parallelism that Fractal uncovers, but they do so at the expense of composability.

Consider the transactional database example introduced earlier. Conventional HTMs run each database transaction in a single thread, and exploit coarse-grained inter-transaction parallelism only. But each database transaction has plentiful ordered parallelism. Fractal can exploit both inter- and intra-transaction parallelism by running each transaction in its own ordered subdomain, just as each global relabel runs in its own ordered subdomain in Figure 4-2. We apply both approaches to the `silo` in-memory database [205]. Figure 4-4 shows that, at 256 cores, `silo-fractal` scales to 206×, while `silo-flat` scales to 9.7× only, 21× slower than `silo-fractal`.

Figure 4-4 also shows that `silo-swarm`, the Swarm version of `silo`, achieves similar performance to `silo-fractal` (`silo-swarm` is 4.5% slower). Figure 4-5 illustrates

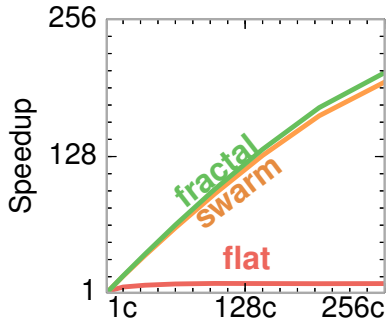


Figure 4-4: Speedup of `silo` versions on 1–256 cores.[1]

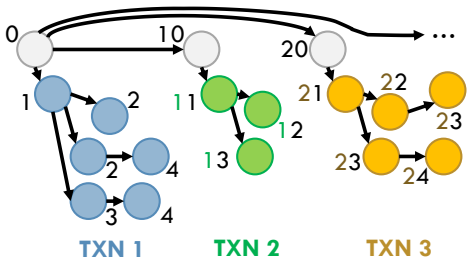


Figure 4-5: `silo-swarm` uses disjoint timestamp ranges for different database transactions, sacrificing composability.

`silo-swarm`'s implementation: the transaction-launching code assigns disjoint timestamp ranges to transactions (10 contiguous timestamps per transaction in Figure 4-5), and each transaction enqueues tasks only within this range (e.g., 10–19 for **TXN 2** in Figure 4-5). `silo-swarm` uses the same fine-grained tasks as `silo-fractal`, exposing plentiful parallelism and reducing the penalty of conflicts [114]. For example, in Figure 4-5, if the tasks at timestamps 13 and 24 conflict, only one task must abort, rather than any whole transaction.

Since Swarm does not provide architectural support for nested parallelism, approaching Fractal's performance comes at the expense of composability. `silo-swarm` *couples* the transaction-launching code and the code within each transaction: both modules must know the number of tasks per transaction, so that they can agree on the semantics of each timestamp. Moreover, a fixed-size timestamp makes it hard to allocate sufficient timestamp ranges in complex applications with many nesting levels or where the number of tasks in each level is dynamically determined. Fractal avoids these issues by providing direct support for nested parallelism.

Prior HTMs have supported composable nested parallel transactions, but they suffer from deadlock and livelock conditions, impose large overheads, and sacrifice most of the benefits of fine-grained parallelism because each nested transaction merges its speculative state with its parent's [19, 20]. We compare Fractal and parallel nesting HTMs in detail in Section 4.6, after discussing Fractal's implementation. Beyond these issues, parallel nesting HTMs do not support ordered parallelism, so they would not help `maxflow` or `silo`.

4.1.3 Fractal Avoids Over-serialization

Beyond forgoing composability, supporting fine-grained parallelism through manually-specified ordering can cause over-serialization.

[1]The speedups here vary marginally from Chapter 3 due to a slightly different system configuration.

Consider the maximal independent set algorithm (`mis`), which, given a graph, finds a set of nodes S such that no two nodes in S are adjacent, and each node not in S is adjacent to some node in S.

`mis` can be easily parallelized with unordered, atomic tasks [186]. We call this implementation `mis-flat`. Each task operates on a node and its neighbors. If the node has not yet been visited, the task visits both the node and its neighbors, adding the node to the set and marking its neighbors as excluded from the set. `mis-flat` creates one task for each node in the graph, and finishes when all these tasks have executed. Figure 4-6 shows that, on an R-MAT graph with 8 million nodes and 168 million edges, `mis-flat` scales to 98× at 256 cores.

`mis-flat` misses a source of nested parallelism: when a node is added to the set, its neighbors may be visited and excluded in parallel. This yields great benefits when nodes have many neighbors. `mis-fractal` defines two task types: include and exclude. An include task checks whether a node has already been visited. If it has not, it adds the node to the set and creates an unordered subdomain to run exclude tasks for the node's neighbors. An exclude task permanently excludes a node from the set. Domains guarantee a node and its neighbors are visited atomically while allowing many tasks of both types to run in parallel. Figure 4-6 shows that `mis-fractal` scales to 145× at 256 cores, 48% faster than `mis-flat`.

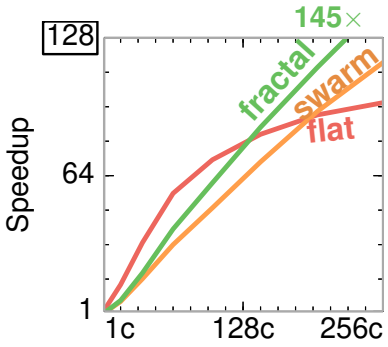


Figure 4-6: Speedup of different `mis` versions on 1–256 cores.

Swarm cannot exploit this parallelism as effectively. Swarm can only guarantee atomicity for groups of tasks if the program specifies a total order among groups (as in `silo`). We follow this approach to implement `mis-swarm`: every include task is assigned a unique timestamp, and it shares its timestamp with any exclude tasks it enqueues. This imposes more order constraints than `mis-fractal`, where there is no order among tasks in the root domain. Figure 4-6 shows that `mis-swarm` scales to 117×, 24% slower than `mis-fractal`, as unnecessary order constraints cause more aborted work.[2]

In summary, conveying the atomicity needs of nested parallelism through a fixed order limits parallel execution. Fractal allows programs to convey nested parallelism without undue order constraints.

4.2 Fractal Execution Model

Figure 4-7 illustrates the key elements of the Fractal execution model. Fractal programs consist of *tasks* in a hierarchy of nested *domains*. Each task may access arbitrary data, and may create child tasks as it finds new work to do. For example, in Figure 4-7 task C creates children D and E. When each task is created, it is enqueued to a specific domain.

[2] `mis-swarm`'s order constraints make it deterministic, which some users may find desirable [25, 64].

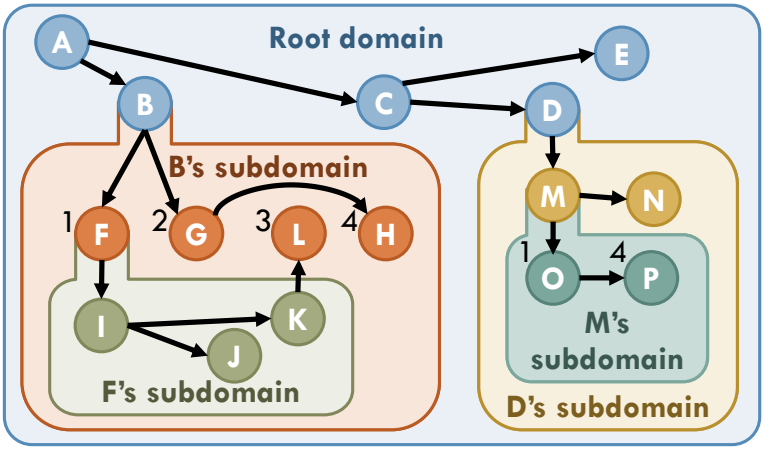


Figure 4-7: Elements of the Fractal execution model. Arrows point from parent to child tasks. Parents enqueue their children into ordered domains where tasks have timestamps, such as A's and M's subdomains, or unordered domains, such as the other three domains.

Semantics within a domain: Each domain provides either unordered or timestamp-ordered execution semantics. In an unordered domain, Fractal chooses an arbitrary order among tasks that respects parent-child dependences, i.e., children are ordered after their parents. For example, in Figure 4-7, task C's children D and E must appear to run after C, but task D can appear to run either before or after task E. These semantics are similar to TM's: all tasks execute atomically and in isolation.

In an ordered domain, each task has a program-specified timestamp. A task can enqueue child tasks to the same domain with any timestamp equal to or greater than its own. Fractal guarantees that tasks appear to run in increasing timestamp order. If multiple tasks have the same timestamp, Fractal arbitrarily chooses an order among them. This order always respects parent-child dependences. Timestamps let programs convey their specific order requirements, e.g., the order that events need to run in a simulator. For example, in Figure 4-7, the timestamps of tasks F, G, L, and H ensure they appear to run in that fixed order. These semantics are the same as Swarm's [114].

Semantics across domains: Each task can create a single subdomain and enqueue tasks into it. For example, in Figure 4-7, task B creates a new subdomain and enqueues F and G into it. These tasks may themselves create their own subdomains. For example, F creates a subdomain and enqueues I into it.

Fractal provides strong atomicity guarantees across domains to allow parallel composition of speculative algorithms. All tasks in a domain appear to execute after the task that creates the domain and are not interleaved with tasks outside their domain. In other words, any non-root domain together with its creator appears to execute as a *single atomic unit* in isolation. For example, since F is ordered before G in B's subdomain, all tasks in F's subdomain (I, J, and K) must appear to execute immediately after F and before G. Furthermore,

```
void exclude(Node& n) {
  n.state = EXCLUDED;
}

void include(Node& n) {
  if (n.state == UNVISITED) {
    n.state = INCLUDED;
    fractal::create_subdomain(UNORDERED);
    for (Node& ngh: n.neighbors)
      fractal::enqueue_sub(exclude, ngh);
  }
}
```

Listing 4.1: Fractal implementation of `mis` tasks.

```
void include(Node& n) {
  if (n.state == UNVISITED) {
    n.state = INCLUDED;
    forall (Node& ngh: n.neighbors)
      ngh.state = EXCLUDED;
  }
}
```

Listing 4.2: Pseudocode for Fractal implementation of `mis`'s `include` using the high-level interface.

although no task in B's subdomain is ordered with respect to any task in D's subdomain, tasks in B's and D's subdomains are guaranteed not to be interleaved.

Tasks may also enqueue child tasks to their immediate enclosing domain, or *superdomain*. For example, in Figure 4-7, K in F's subdomain enqueues L to B's subdomain. This lets a task delegate enqueuing future work to descendants within the subdomain it creates. A task cannot enqueue children to any domain beyond the domain it belongs to, its superdomain, and the single subdomain it may create.

4.2.1 Programming Interface

We first expose Fractal's features through a simple low-level C++ interface, then complement it with a high-level, OpenMP-style interface that makes it easier to write Fractal applications.

Low-level interface: Listing 4.1 illustrates the key features of the low-level Fractal interface by showing the implementation of the `mis-fractal` tasks described in Section 4.1.3. A task is described by its function, arguments, and ordering properties. Task functions can take arbitrary arguments but do not return values. Tasks create children by calling one of three enqueue functions with the appropriate task function and arguments: `fractal::enqueue` places the child task in the same domain as the caller, `fractal::enqueue_sub` places the child in the caller's subdomain, and `fractal::enqueue_super` places the child in the caller's superdomain. If the destination domain is ordered, the enqueuing function also takes the child task's timestamp. This isn't the case in Listing 4.1, as `mis` is unordered.

Before calling `fractal::enqueue_sub` to place tasks in a subdomain, a task must call `fractal::create_subdomain` exactly once to specify the subdomain's ordering semantics: unordered, or ordered with 32- or 64-bit timestamps. In Listing 4.1, each `include` task may create an unordered subdomain to atomically run `exclude` tasks for all its neighbors. The initialization code (not shown) creates an `include` task for every node in an unordered root domain.

Task enqueue functions also take one optional argument, a spatial hint. Hints aid the system in performing locality-aware task mapping and load balancing. Hints are orthogonal to Fractal. We adopt them because we study systems of up to 256 cores, and several of our benchmarks suffer from poor locality without hints, which limits their scalability beyond tens of cores.

High-level interface: Although our low-level interface is simple, breaking straight-line code into many task functions can be tedious. To ease this burden, we implement a high-level interface in the style of OpenMP and OpenTM [21]. Table 4.1 details its main constructs, and Listing 4.2 shows it in action with *pseudocode* for `include`. Nested parallelism is expressed using `forall`, which automatically creates an unordered subdomain and enqueues each loop iteration as a separate task. This avoids breaking code into small functions like `exclude`. These constructs can be arbitrarily nested. Our actual syntax is slightly more complicated because we do not modify the compiler, and we implement these constructs using macros.[3]

4.3 Fractal Implementation

Our Fractal implementation seeks three desirable properties. First, the architecture should perform *fine-grained speculation*, carrying out conflict resolution and commits at the level of individual tasks, not complete domains. This avoids the granularity issues of nested parallel HTMs (Section 4.6). Second, *creating a domain should be cheap*, as domains with few tasks are common (e.g., `mis` in Section 4.1.3). Third, while the architecture should support unbounded nesting depth to enable software composition, parallelism compounds quickly with depth, so *hardware only needs to support a few concurrent depths*.

To meet these objectives, our Fractal implementation builds on Swarm, and dynamically chooses a task commit order that satisfies Fractal's semantics. We first briefly recap the Swarm microarchitecture, then introduce the modifications needed to support Fractal.

The baseline Swarm microarchitecture is as described in Section 3.3. We have cache-coherent tiled multicore (Figure 3-2). Each tile has a group of simple 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

[3] The difference between the pseudocode in Listing 4.2 and our actual code is that we have to tag the end of control blocks, i.e., using `forall_begin(...) {...} forall_end();`. This could be avoided with compiler support, as in OpenMP.

Function	Description
`forall`	Atomic unordred loop. Enqueues each iteration as as a task in a new unordered subdomain.
`forall_ordered`	Atomic ordered loop. Enqueues tasks to a new ordered subdomain, using the iteration index as a timestamp.
`forall_reduce`	Atomic unordered loop with a reduction variable.
`forall_reduce_ordered`	Atomic ordered loop with a reduction variable.
`parallel`	Execute multiple code blocks as parallel tasks.
`parallel_reduce`	Execute multiple code blocks as parallel tasks, followed by a reduction.
`enqueue_all`	Enqueues a sequence of tasks with the same (or no) timestamp.
`enqueue_all_ordered`	Enqueues a sequence of tasks with a range of timestamps.
`task`	Starts a new task in the middle of a function. Implicitly encapsulates the rest of the function into a lambda, then enqueues it. Useful to break long functions into smaller tasks.
`callcc`	Call with current continuation [199]. Allows calling a function that might enqueue tasks, returning control to the caller by invoking its continuation. The continuation runs as a separate task.

Table 4.1: High-level interface functions.

commits tasks. The architecture executes tasks speculatively and out of order to scale. We incorporate the low-overhead hardware task management, large task queues, scalable data-dependence speculation techniques, and high-throughput ordered commits as described in Section 3.3.

Swarm maintains a consistent order among tasks by giving a unique *virtual time* (VT) to each task when it is dispatched. Swarm VTs are 128-bit integers that extend the 64-bit program-assigned timestamp with a 64-bit *tiebreaker*. This tiebreaker is the concatenation of the *dispatch cycle* and *tile id*, as shown in Figure 4-8. Thus, Swarm VTs break ties among same-timestamp tasks sensibly (prioritizing older tasks), and they satisfy Swarm's semantics (they order a child task after its parent, since the child is always dispatched at a later cycle). However, Fractal needs a different schema to match its semantics.

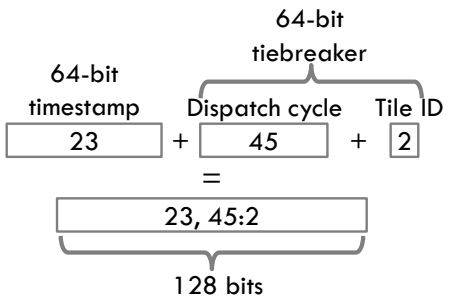


Figure 4-8: Swarm VT construction.

4.3.1 Fractal Virtual Time

Fractal assigns a *fractal virtual time* (fractal VT) to each task. This fractal VT is the concatenation of one or more *domain virtual times* (domain VTs).

Domain VTs order all tasks in a domain and are constructed similarly to Swarm VTs. In an ordered domain, each task's domain VT is the concatenation of its 32- or 64-bit timestamp and a tiebreaker. In an unordered domain, tasks do not have timestamps, so each task's domain VT is just a tiebreaker, assigned at dispatch time.

Fractal uses 32-bit rather than 64-bit tiebreakers for efficiency. As in Swarm, each tiebreaker is the concatenation of *dispatch cycle* and *tile id*, which orders parents before children. While 32-bit tiebreakers are efficient, they can wrap around. Section 4.3.3 discusses how Fractal handles wrap-arounds. Figure 4-9 illustrates the possible formats of a domain VT, which can take 32, 64, or 96 bits.

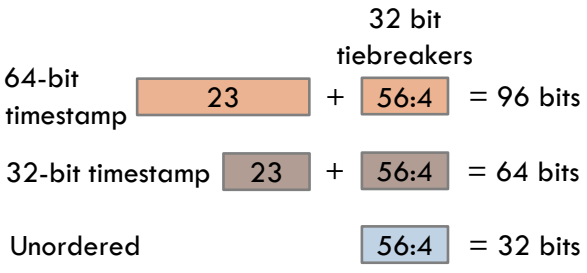


Figure 4-9: Domain VT formats.

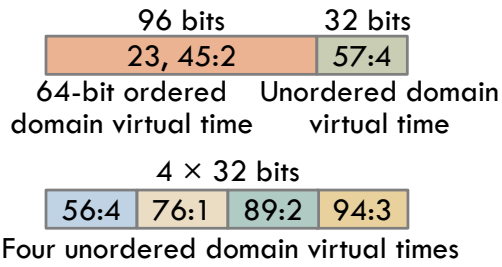


Figure 4-10: Example 128-bit fractal VTs.

Fractal VTs enforce a total order among tasks in the system. This order satisfies Fractal's semantics across domains: all tasks within each domain are ordered immediately after the domain's creator and before any other tasks outside the domain. These semantics can be implemented with two simple rules. First, the fractal VT of a task in the root domain is just its root domain VT. Second, the fractal VT of any other task is equal to its domain VT appended to the fractal VT of the task that created its domain. Figure 4-10 shows some example fractal VT formats. A task's fractal VT is thus made up of one domain VT for each enclosing domain. Two fractal VTs can be compared with a natural lexicographic comparison.

Fractal VTs are easy to support in hardware. We use a fixed-width field in the task descriptor to store each fractal VT, 128 bits in our implementation. Fractal VTs smaller than 128 bits are right-padded with zeros. This fixed-width format makes comparing fractal VTs easy, requiring conventional 128-bit comparators. With a 128-bit budget, Fractal hardware can support up to four levels of nesting, depending on the sizes of domain VTs. Section 4.3.2 describes how to support levels beyond those that can be represented in 128 bits.

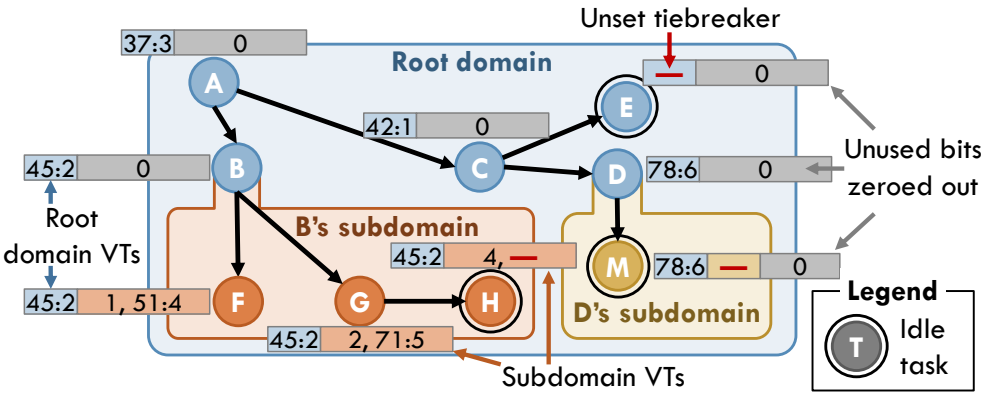


Figure 4-11: Fractal VTs in action.

Figure 4-11 shows fractal VTs in a system with three domains: an unordered root domain, B's subdomain (ordered with 64-bit timestamps), and D's subdomain (unordered). Idle tasks do not have tiebreakers, which are assigned on dispatch. Any two dispatched tasks can be ordered by comparing their fractal VTs. For example, F (in B's subdomain) is ordered after B, but before M (in D's subdomain). Fractal performs fine-grained speculation by using the total order among running tasks to commit and resolve conflicts at the level of individual tasks. For example, although all tasks in B's subdomain must stay atomic with respect to tasks in any other domain, Fractal can commit tasks B and F individually, without waiting for G and H to finish. Fractal guarantees that B's subdomain executes atomically because G and H are ordered before any of the remaining uncommitted tasks.

Fractal VTs also make it trivial to create a new domain. In hardware, enqueuing to a subdomain simply requires including the parent's full fractal VT in the child's task descriptor. For instance, when B enqueues F in Figure 4-11, it tags F with (45:2; 1)—B's fractal VT (45:2) followed by F's timestamp (1). Similarly, enqueues to the same domain use the enqueuer's fractal VT without its final domain VT (e.g., when A enqueues C, C's fractal VT uses no more bits than A's), and enqueues to the superdomain use the enqueuer's fractal VT without its final two domain VTs.

In summary, fractal VTs capture all the information needed for ordering and task enqueues, so these operations do not rely on centralized structures. Moreover, the rules of fractal VT construction automatically enforce Fractal's semantics across domains while performing speculation only at the level of fine-grained tasks—no tracking is done at the level of whole domains.

4.3.2 Supporting Unbounded Nesting

Large applications may consist of parallel algorithms nested with arbitrary depth. Fractal supports this unbounded nesting depth by spilling tasks from shallower domains to memory. These spilled tasks are filled back into the system after deeper domains finish. This

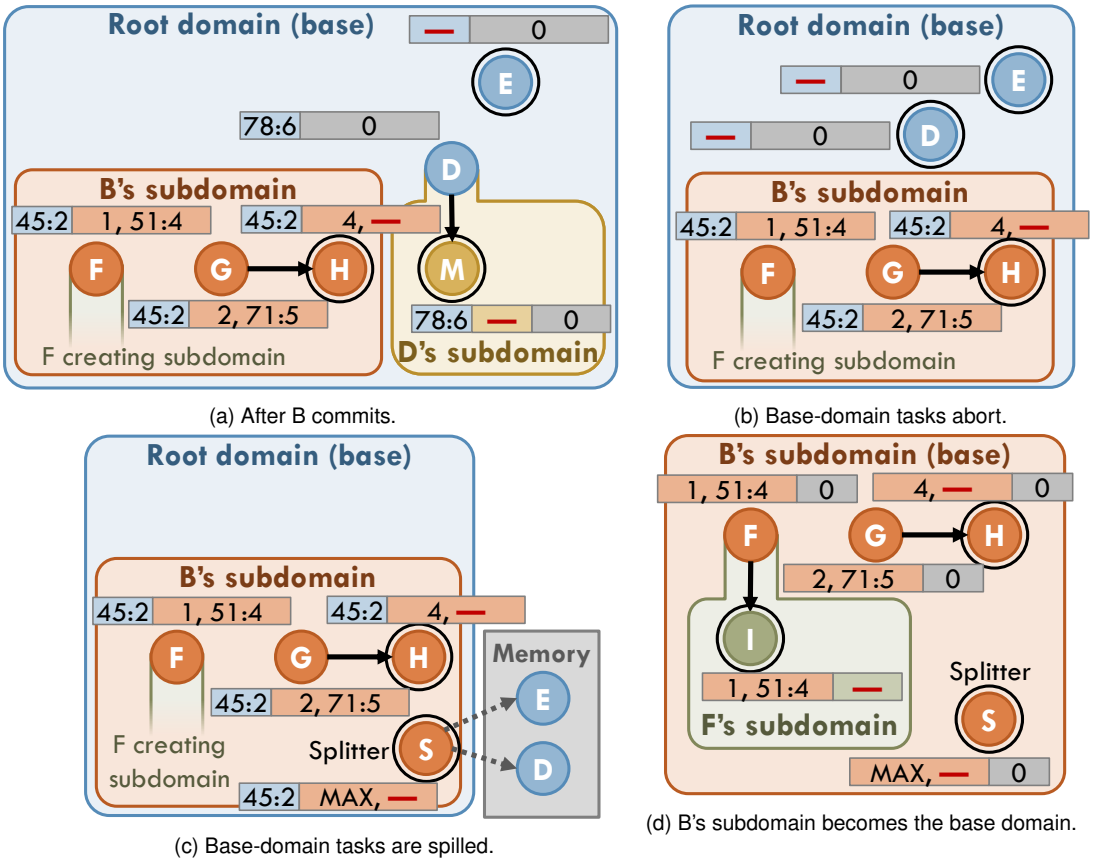


Figure 4-12: Starting from Figure 4-11, zooming in allows F to create and enqueue to a subdomain by shifting fractal VTs.

process, which we call *zooming*, is conceptually similar to the stack spill-fill mechanism in architectures with register windows [93]. Zooming in allows Fractal to continue fine-grained speculation among tasks in deeper domains, without requiring additional structures to track speculative state. Note that, although zooming is involved, it imposes negligible overheads: zooming is not needed in our full applications (which use two nesting levels), and it happens infrequently in microbenchmarks (Section 4.5.3).

Zooming in spills tasks from the shallowest active domain, which we call the *base domain*, to make space for deeper domains. Suppose that, in Figure 4-11, F in B's subdomain wants to create an unordered subdomain and enqueue a child into it. The child's fractal VT must include a new subdomain VT, but no bits are available to the right of F's fractal VT. To solve this, F issues a *zoom-in request* to the central arbiter with its fractal VT.

Figure 4-12 illustrates the actions taken during a zoom-in. To avoid priority inversion, the task that requests the zoom-in waits until the base domain task that shares its base domain VT commits. This guarantees that no active base domain tasks precede the requesting task. In our example, F waits until B commits. Figure 4-12a shows the state of the sys-

tem at this point—note that F and all other tasks in B's subdomain precede the remaining base-domain tasks. The arbiter broadcasts the zoom-in request and saves any timestamp component of the base domain VT to an in-memory stack. In Figure 4-12a, the base domain is unordered so there is no timestamp for the arbiter to save.

Each zoom-in proceeds in two steps. First, all tasks in the base domain are spilled to memory. For simplicity, speculative state is never spilled. Instead, any base-domain tasks that are running or have finished are aborted first, which recursively aborts and eliminates their descendants. Figure 4-12b shows the state of the system after these aborts. Note how D's abort eliminates M and D's entire subdomain. Although spilling tasks to memory is complex, it reuses the spilling mechanism already present in Swarm [114]: task units dispatch *coalescer* tasks that remove base-domain tasks from task queues, store them in memory, and enqueue a *splitter* task that will later re-enqueue the spilled tasks. The splitter task is deprioritized relative to all regular tasks. Figure 4-12c shows the state of the system once all base-domain tasks have been spilled. A new splitter task, S, will re-enqueue D and E to the root domain when it runs.

In the second step of zooming in, the system turns the outermost subdomain into the base domain. At this point, all tasks belong to one subdomain (B's subdomain in our example), so their fractal VTs all begin with the same base domain VT. This common prefix may be eliminated while preserving order relations. Each tile walks its task queues and modifies the fractal VTs of all tasks by shifting out the common base domain VT. Each tile also modifies its canary VTs, which enable the L2 to filter conflict checks [114]. Overall, this requires modifying a few tens to hundreds of fractal VTs per tile (in our implementation, up to 256 in the task queue and up to 128 canaries). Figure 4-12d shows the state of the system after zooming in. B's subdomain has become the base domain. This process has freed 32 bits of fractal VT, so F can enqueue I into its new subdomain.

Zooming out reverses the effects of zooming in. It is triggered when a task in the base domain wants to enqueue to its superdomain. The enqueuing task first waits until all tasks preceding it have committed. Then, it sends a *zoom-out request* to the central arbiter with its fractal VT. If the previous base domain was ordered, the central arbiter pops a timestamp from its stack to broadcast with the zoom-out request.

Zooming out restores the previous base domain: Each tile walks its task queues, right-shifting each fractal VT and adding back the base domain timestamp, if any. The restored base domain VT has its tiebreaker set to zero, but this does not change any order relations because the domain from which we are zooming out contains all the earliest active tasks.

Avoiding quiescence: As explained so far, the system would have to be completely quiesced while fractal VTs are being shifted. This overhead is small—a few hundred cycles—but introducing mechanisms to quiesce the whole system would add complexity. Instead, we use an alternating-bit protocol [200] to let tasks continue running while fractal VTs are modified. Each fractal VT entry in the system has an extra bit that is flipped on each zoom

in/out operation. When the bits of two fractal VTs being compared differ, one of them is shifted appropriately to perform the comparison.

4.3.3 Handling Tiebreaker Wrap-arounds

Using 32-bit tiebreakers makes fractal VTs compact, but causes tiebreakers to wrap around every few tens of milliseconds. Since domains can exist for long periods of time, the range of existing tiebreakers must be compacted to make room for new ones. When tiebreakers are about to wrap around, the system walks every fractal VT and performs the following actions:

1. Subtract 2^{31} (half the range) with saturate-to-0 from each tiebreaker in the fractal VT (i.e., flip the MSB from 1 to 0, or zero all the bits if the MSB was 0).

2. If a task's *final* tiebreaker is 0 after subtraction *and* the task is not the earliest unfinished task, abort it.

When this process finishes, all tiebreakers are $< 2^{31}$, so the system continues assigning tiebreakers from 2^{31}.

This exploits the property that, if the task that created a domain precedes all other active tasks, its tiebreaker can be set to zero without affecting order relations. If the task is aborted because its tiebreaker is set to 0, any subdomain it created will be squashed. In practice, we find this has no effect on performance, because, to be aborted, a task would have to remain speculative for far longer than we observe in any benchmark.

4.3.4 Putting It All Together

Our Fractal implementation adds small hardware overheads over Swarm. (Swarm itself imposes modest overheads to implement speculative execution [114].) Each fractal VT consumes five additional bits beyond Swarm's 128: four to encode its format (14 possibilities), and one for the alternating-bit protocol. This adds storage overheads of 240 bytes per 4-core tile. Fractal also adds simple logic to each tile to walk and modify fractal VTs—for zooming and tiebreaker wrap-arounds—and adds a shifter to fractal VT comparators to handle the alternating-bit protocol.

Fractal makes small changes to the ISA: it modifies the `enqueue` instruction and adds a `create_subdomain` instruction. Task enqueue messages carry a fractal VT without the final tiebreaker (up to 96+5 bits) compared to the 64-bit timestamp in Swarm.

Finally, in our implementation, zoom-in/out requests and tiebreaker wrap-arounds are handled by the global virtual time arbiter (the unit that runs the ordered-commit protocol). This adds a few message types between this arbiter and the tiles to carry out the steps in each of these operations. The arbiter must manage a simple in-memory stack to save and restore base domain timestamps.

4.4 Experimental Methodology

Modeled system: We use the same methodology as the previous chapter (Section 3.4). We model Fractal systems of up to 256 cores, as shown in Figure 3-2, with parameters in Table 4.2. Swarm parameters (task and commit queue sizes, etc.) match those in Table 3.3. Similar to Section 3.4, we keep *per-core* L2/L3 sizes and queue capacities constant across system sizes. This captures performance per unit area. As a result, larger systems have higher queue and cache capacities, which sometimes cause superlinear speedups.

Fractal instructions	5 cycles per `enqueue/dequeue/finish_task` 2 cycles per `create_subdomain` instruction
Fractal VT	128 bits per task descriptor

Table 4.2: Configuration of the 256-core Fractal system. Core, memory system, and Swarm parameters as listed in Table 3.3.

	Application	Input	1-core runtime (B cycles)
Swarm	**color** [98]	com-youtube [126]	0.968
	msf [186]	kron_g500-logn16 [18, 61]	0.717
	silo [205]	TPC-C, 4 whs, 32 Ktxns	2.98
STAMP [134]	**ssca2**	-s15 -i1.0 -u1.0 -l6 -p6	10.6
	vacation	-n4 -q60 -u90 -r1048576 -t262144	4.31
	genome	-g4096 -s48 -n1048576	2.26
	kmeans	-m40 -n40 -i rand-n16384-d24-c16	8.75
	intruder	-a10 -l64 -s32768	2.12
	yada	-a15 -i ttimeu100000.2	3.41
	labyrinth	random-x128-y128-z5-n128	4.41
	bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	8.81
	maxflow [22]	rmf-wide [50, 88], 65 K nodes, 314 K edges	16.7
	mis [186]	R-MAT [43], 8 M nodes, 168 M edges	1.34

Table 4.3: Benchmark information: source implementations, inputs, and execution time on a single-core system.

Benchmarks: Table 4.3 reports the benchmarks we evaluate. Benchmarks have 1-core run-times of about 1 B cycles or longer. We use three existing Swarm benchmarks, which we adapt to Fractal; Fractal implementations of the eight STAMP benchmarks [134]; and two new Fractal benchmarks: `maxflow`, adapted from `prsn` [22], and `mis`, adapted from PBBS [186].

Benchmarks adapted from Swarm use their same inputs. `msf` includes an optimization to filter out non-spanning edges efficiently [25]. This optimization improves absolute per-

formance but reduces the amount of highly parallel work, so `msf` has lower scalability than the unoptimized Swarm version [114].

STAMP benchmarks use inputs between the recommended "+" and "++" sizes, to achieve a run-time large enough to evaluate 256-core systems, yet small enough to be simulated in reasonable time.

`maxflow` uses rmf-wide [88], one of the harder graph families from the DIMACS maxflow challenge [22]. `mis` uses an R-MAT graph [43], which has a power-law distribution.

We fast-forward each benchmark to the start of its parallel region (skipping initialization), and report results for the full parallel region. On all benchmarks except `bayes`, we perform enough runs to achieve 95% confidence intervals $\leq 1\%$. `bayes` is highly non-deterministic, so we report its average results with 95% confidence intervals over 50 runs.

4.5 Evaluation

We now analyze the benefits of Fractal in depth. As in Section 4.1, we begin with applications where Fractal uncovers abundant fine-grained parallelism through nesting. We then discuss Fractal's benefits from avoiding over-serialization. Finally, we characterize the performance overheads of zooming to support deeper nesting.

	Perf. vs serial @ 1-core		**Avg task length (cycles)**		**Nesting type**
	`flat`	`fractal`	`flat`	`fractal`	
maxflow	0.92×	0.68×	3260	373	unord ↪ ord-32b
labyrinth	1×	0.62×	16 M	220	unord ↪ ord-32b
bayes	1×	1.11×	1.8 M	3590	unord ↪ unord
silo	1.14×	1.10×	80 K	3420	unord ↪ ord-32b
mis	0.79×	0.26×	162	115	unord ↪ unord
color	1.06×	0.80×	633	96	ord-32b ↪ ord-32b
msf	3.1×	1.73×	113	49	ord-64b ↪ unord

Table 4.4: Benchmarks with parallel nesting: performance of 1-core `flat`/`fractal` vs tuned serial versions (higher is better), average task lengths, and nesting semantics.

4.5.1 Fractal Uncovers Abundant Parallelism

Fractal's support for nested parallelism greatly benefits three benchmarks: `maxflow`, as well as `labyrinth` and `bayes`, the two least scalable benchmarks from STAMP.

maxflow , as discussed in Section 4.1.1, is limited by long global-relabel tasks. Our `fractal` version performs the breadth-first search nested within each global relabel in parallel.

labyrinth finds non-overlapping paths between pairs of *(start, end)* cells on a 3D grid. Each transaction operates on one pair: it finds the shortest path on the grid and claims the cells on the path for itself. In the STAMP implementation, each transaction performs this shortest-path search sequentially. Our `fractal` version runs the shortest-path search nested within each transaction in parallel, using an ordered subdomain.

bayes learns the structure of a Bayesian network, a DAG where nodes denote random variables and edges denote conditional dependencies among variables. `bayes` spends most time deciding whether to insert, remove, or reverse network edges. Evaluating each decision requires performing many queries to an ADTree data structure, which efficiently represents probability estimates. In the STAMP implementation, each transaction evaluates and applies an insert/remove/reverse decision. Since the ADTree queries performed depend on the structure of the network, transactions serialize often. Our `fractal` version runs ADTree queries nested within each transaction in parallel, using an unordered subdomain.

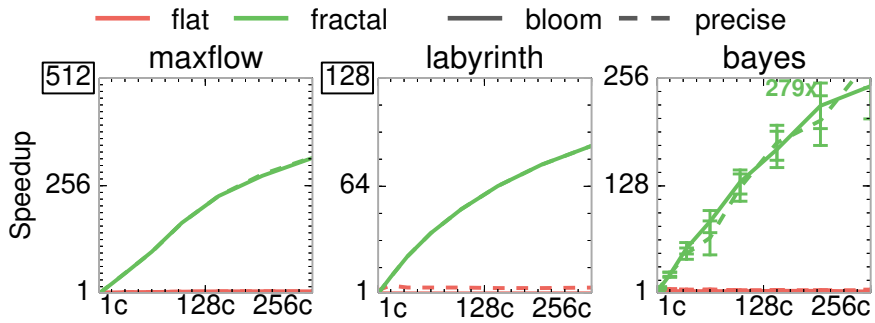
Table 4.4 compares the 1-core performance and average task lengths of `flat` and `fractal` versions. `flat` versions of these benchmarks have long, unordered transactions (up to 16 M cycles). `fractal` versions have much smaller tasks (up to 3590 cycles on average in `bayes`). These short tasks hurt serial performance (by up to 38% in `labyrinth`), but expose plentiful *intra-domain parallelism* (e.g., a parallel breadth-first search), yielding great scalability.

Beyond limiting parallelism, the long transactions of `flat` versions have large read-/write sets that often overflow Fractal's Bloom filters, causing false-positive aborts. Therefore, we also present results under an idealized, *precise* conflict detection scheme that does not incur false positives. High false positive rates are not specific to Fractal—prior HTMs used similarly-sized Bloom filters [42, 135, 180, 219].

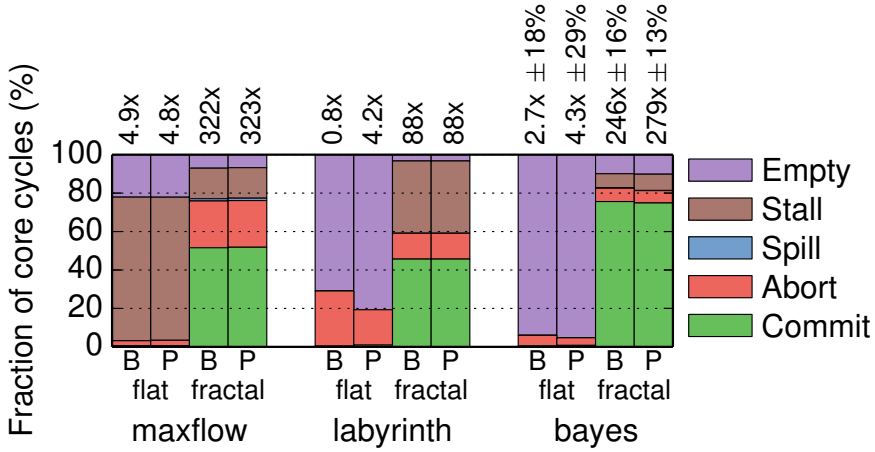
Figure 4-13a shows the performance of the `flat` and `fractal` versions when scaling from 1- to 256-core systems. All speedups reported are over the 1-core `flat` version. Solid lines show speedups when using Bloom filters, while dashed ones show speedups under precise conflict detection. `flat` versions scale poorly, especially when using Bloom filters: the maximum speedups across all system sizes range from 1.0× (`labyrinth` at 1 core) to 4.9× (`maxflow`). By contrast, `fractal` versions scale much better, from 88× (`labyrinth`) to 322× (`maxflow`).[4]

Figure 4-13b gives more insight into these differences by showing the percentage of cycles that cores spend on different activities: *(i)* running tasks that are ultimately com-

[4] Note that systems with more tiles have higher cache and queue capacities, which sometimes cause superlinear speedups (Section 4.4).



(a) Speedup from 1 to 256 cores relative to 1-core `flat`.



(b) Breakdown of core cycles at 256 cores, with speedups on top.

Figure 4-13: Performance of `flat` and `fractal` versions of applications with abundant nested parallelism, using Bloom filter–based or Precise conflict detection.

mitted, *(ii)* running tasks that are later aborted, *(iii)* spilling tasks from the hardware task queues, *(iv)* stalled on a full task or commit queue, or *(v)* stalled due to lack of tasks. Each group of bars shows results for a different application at 256 cores.

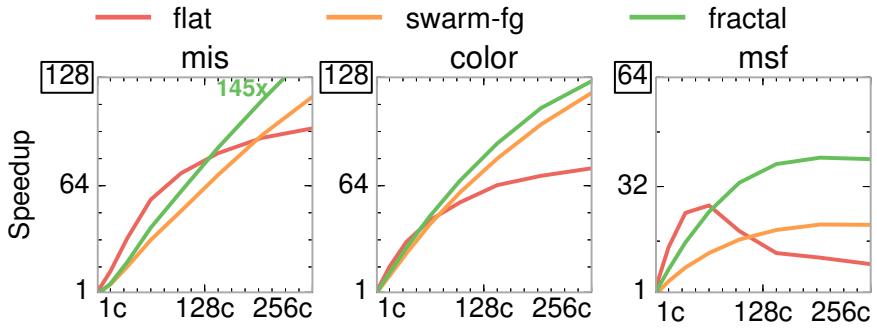
Figure 4-13b shows that `flat` versions suffer from lack of work caused by insufficient parallelism, and stalls caused by long tasks that eventually become the earliest active task and prevent others from committing. Moreover, most of the work performed by `flat` versions is aborted as tasks have large read/write sets and frequently conflict. `labyrinth-flat` and `bayes-flat` also suffer frequent false-positive aborts that hurt performance with Bloom filter conflict detection. Although precise conflict detection helps `labyrinth-flat` and `bayes-flat`, both benchmarks still scale poorly (to 4.3× and 6.8×, respectively) due to insufficient parallelism.

By contrast, `fractal` versions spend most cycles executing useful work, and aborted cycles are relatively small, from 7% (`bayes`) to 24% (`maxflow`). `fractal` versions perform just as well with Bloom filters as with precise conflict detection. These results shows that exploiting fine-grained nested speculative parallelism is an effective way to scale challenging applications.

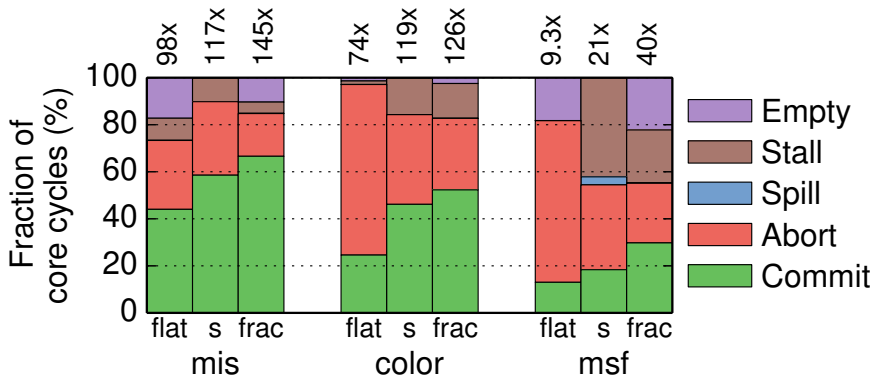
4.5.2 Fractal Avoids Over-serialization

Fractal's support for nested parallelism avoids over-serialization on four benchmarks: `silo`, `mis`, `color`, and `msf`. Swarm can exploit nested parallelism in these benchmarks by imposing a total order among coarse-grained operations or groups of tasks (Section 4.1.3). Section 4.1 showed that this has a negligible effect on `silo`, so we focus on the other three applications.

`mis`, `color`, and `msf` are graph-processing applications. Their `flat` versions perform operations on multiple graph nodes that can be parallelized but must remain atomic—e.g., in `mis`, adding a node to the independent set and excluding its neighbors (Section 4.1.3). `mis-flat` is unordered, while `color-flat` and `msf-flat` visit nodes in a *partial* order (e.g., `color` visits larger-degree nodes first). Our `fractal` versions use one subdomain per coarse-grained operation to exploit this nested parallelism (Table 4.4). The `swarm-fg` versions of these benchmarks use the same fine-grained tasks as `fractal` but use a unique timestamp or timestamp range per coarse-grained operation to guarantee atomicity, imposing a *fixed* order among coarse-grained operations.



(a) Speedup from 1 to 256 cores relative to 1-core `flat`.



(b) Breakdown of core cycles at 256 cores, with speedups on top.

Figure 4-14: Performance of `flat`, `swarm-fg`, and `fractal` versions of applications where Swarm extracts nested parallelism through strict ordering, but Fractal outperforms it by avoiding undue serialization.

Figure 4-14 shows the scalability and cycle breakdowns for these benchmarks. `flat` versions achieve the lowest speedups, from 26× (`msf` at 64 cores) to 98× (`mis`). Fig-

ure 4-14b shows that they are dominated by aborts, which take up to 73% of cycles in `color-flat`, and empty cycles caused by insufficient parallelism in `msf` and `mis`. In `msf-flat`, frequent aborts hurt performance beyond 64 cores.

By contrast, `fractal` versions achieve the highest performance, from 40× (`msf`) to 145× (`mis`). At 256 cores, the majority of time is spent on committed work, although aborts are still noticeable (up to 30% of cycles in `color`). While `fractal` versions perform better at 256 cores, their tiny tasks impose higher overheads, so they underperform `flat` on small core counts. This is most apparent in `msf`, where `fractal` tasks are just 49 cycles on average (Table 4.4).

Finally, `swarm-fg` versions follow the same scaling trends as `fractal` ones, but over-serialization makes them 6% (`color`), 24% (`mis`), and 93% (`msf`) slower. Figure 4-14b shows that these slowdowns primarily stem from more frequent aborts. This is because in `swarm-fg` versions, conflict resolution priority is static (determined by timestamps), while in `fractal` versions, it is based on the dynamic execution order (determined by tiebreakers). In summary, these results show that Fractal makes fine-grained parallelism more attractive by avoiding needless order constraints.

4.5.3 Zooming Overheads

Although our Fractal implementation supports unbounded nesting (Section 4.3.2), two nesting levels suffice for all the benchmarks we evaluate. Larger programs should require deeper nesting. Therefore, we use a microbenchmark to characterize the overheads of Fractal's zooming technique.

Our microbenchmark stresses Fractal by creating many nested domains that contain few tasks each. Specifically, it generates a depth-8 tree of nested domains with fanout F. All tasks perform a small, fixed amount of work (1500 cycles). Non-leaf tasks then create an unordered subdomain and enqueue F children into it. We sweep both the fanout ($F = 4$ to 12) and the maximum number of concurrent levels D in Fractal, from 2 (64-bit fractal VTs) to 8 (256-bit fractal VTs). At $D = 8$, the system does not perform any zooming. Our default hardware configuration supports up to 4 concurrent levels.

Figure 4-15a reports performance on a 1-core system. Each group of bars shows results for a single fanout, and bars within a group show how performance changes as the maximum concurrent levels D grows from 2 to 8. Performance is relative to the $D = 8$, no-zooming system. Using a 1-core system lets us focus on the overheads of zooming without factoring in limited parallelism. Larger fanouts and concurrent levels increase the amount of work executed between zooming operations, reducing overheads. Nonetheless, overheads are modest even for $F = 4$ and $D = 2$ (21% slowdown).

Figure 4-15b reports performance on a 256-core system. Supporting a limited number of levels reduces parallelism, especially with small fanouts, which hurts performance. Nonetheless, as long as $F \geq 8$, supporting at least four levels keeps overheads small.

All of our applications have much higher parallelism than 8 concurrent tasks in at least one of their two nesting levels, and often in both. Therefore, on applications with deeper

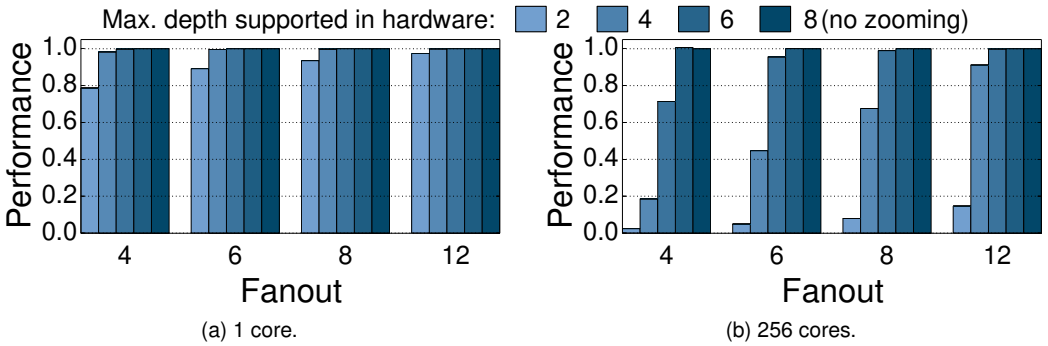


Figure 4-15: Performance of `zoom-tree` microbenchmark for different fanouts F, as we vary the number of concurrent levels D supported in hardware. a) Fractal imposes only modest overheads even for small F, D, b) fanouts ≥ 8, as seen in most of our applications, impose negligible overheads.

nesting, zooming should not limit performance in most cases. However, these are carefully coded applications that avoid unnecessary nesting. Nesting could be overused (e.g., increasing the nesting depth at every intermediate step of a divide-and-conquer algorithm), which would limit parallelism. To avoid this, a compiler pass may be able to safely flatten unnecessary nesting levels. We leave this to future work.

4.5.4 Discussion

We considered 18 benchmarks to evaluate Fractal: all eight from Swarm [113, 114], all eight from STAMP [134], as well as `maxflow` and `mis`. We looked for opportunities to exploit nested parallelism, focusing on benchmarks with limited speedups. In summary, Fractal benefits 7 out of these 18 benchmarks. We did not find opportunities to exploit nested parallelism in the five Swarm benchmarks not presented here (`bfs`, `sssp`, `astar`, `des`, and `nocsim`). These benchmarks already use fine-grained tasks and scale well to 256 cores.

Figure 4-16 shows how each STAMP benchmark scales when using different Fractal features. All speedups reported are over the 1-core **TM** version. The **TM** lines show the performance of the original STAMP transactions ported to Swarm tasks. Three applications (`intruder`, `labyrinth`, and `bayes`) barely scale, while two (`yada` and `kmeans`) scale well at small core counts but suffer on larger systems. By contrast, Fractal's features make all STAMP applications scale, although speedups are not only due to nesting. First, the TM versions of `intruder` and `yada` use software task queues that limit their scalability. Refactoring them to use Swarm/Fractal hardware task queues [114] makes them scale. Second, spatial hints [113] improves `genome` and makes `kmeans` scale. Finally, as we saw in Section 4.5.1, Fractal's support for nesting makes `labyrinth` and `bayes` scale. Therefore, Fractal is the first architecture that scales the full STAMP suite to hundreds of cores, achieving a gmean speedup of 177× at 256 cores.

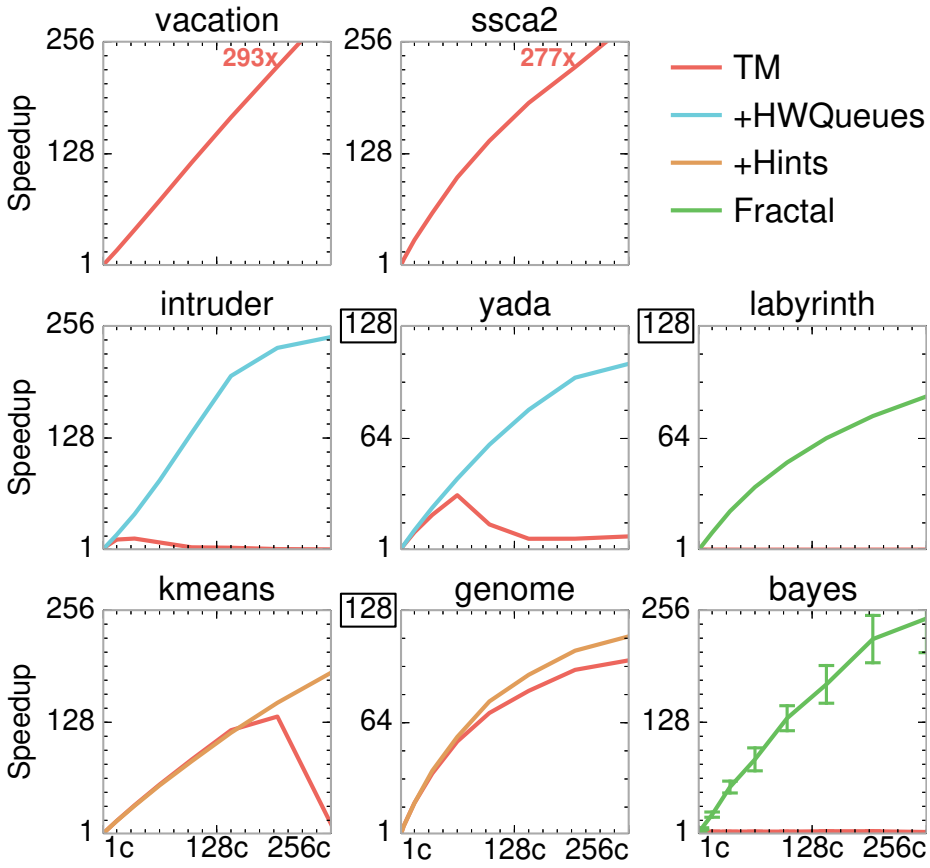


Figure 4-16: Different Fractal features make all STAMP applications scale well to 256 cores.

4.6 Related Work

4.6.1 Nesting in Transactional Memory

Serial nesting: Most HTMs support *serial* execution of nested transactions, which makes transactional code easy to compose but forgoes intra-transaction parallelism. Nesting can be trivially supported by ignoring the boundaries of all nested transactions, treating them as part of the top-level one. Some HTMs exploit nesting to implement *partial aborts* [138]: they track the speculative state of a nested transaction separately while it executes, so conflicts that occur while the nested transaction runs do not abort the top-level one.

Even with partial aborts, HTMs ultimately merge nested speculative state into the top-level transaction, resulting in large atomic regions that are hard to support in hardware [11, 31, 53, 54] and make conflicts more likely.

Prior work has explored relaxed nesting semantics, like open nesting [133, 138, 144] and early release [188], which relax isolation to improve performance. Fractal is orthogonal to these techniques and could be extended to support them, but we do not see the need on the applications we study.

Parallel nesting: Some TM systems support running nested transactions in parallel [140]: a transaction can launch multiple nested transactions and wait for them to finish. Nested transactions may run in parallel and can observe updates from their parent transaction. As in serial nesting, when a nested transaction finishes, its speculative state is merged with its parent's. When all nested transactions finish, the parent transaction resumes execution.

Most of this work has been in software TM (STM) implementations [5, 19, 65, 209], but these suffer from even higher overheads than flat STMs. Vachharajani [206, Ch. 7] and FaNTM [20] introduce hardware support to reduce parallel nesting overheads. Even with hardware support, parallel-nesting HTMs yield limited gains—e.g., FaNTM is often slower than a flat HTM, and moderately outperforms it (by up to 40%) only on a microbenchmark.

Parallel-nesting TMs suffer from three main problems. First, nested transactions merge their speculative state with their parent's, and only the coarse, top-level transaction can commit. This results in large atomic blocks that are as expensive to track and as prone to abort as large serial transactions. By contrast, Fractal performs fine-grained speculation, at the level of individual tasks. It never merges the speculative state of tasks, and relies on ordering tasks to guarantee the atomicity of nested domains.

Second, because the parent transaction waits for its nested transactions to finish, there is a cyclic dependence between the parent and its nested transactions. This introduces many subtle problems, including data races with the parent, deadlock, and livelock [20]. Workarounds for these issues are complex and sacrifice performance (e.g., a nested transaction eventually aborts all its ancestors for livelock avoidance [20]). By contrast, all dependences in Fractal are acyclic, from parents to children, which avoids these issues. Fractal supports the fork-join semantics of parallel-nesting TMs by having nested transactions enqueue their parent's continuation.

Finally, parallel-nesting TMs do not support ordered speculative parallelism. By contrast, Fractal supports arbitrary nesting of ordered and unordered parallelism, which accelerates a broader range of applications.

4.6.2 Thread-Level Speculation

A few TLS systems use timestamps internally, but do not let programs control them [95, 171, 195]. Renau et al. [171] use timestamps to allow out-of-order task spawn. Each task carries a timestamp range, and splits it in half when it spawns a successor. This approach could be adapted to support the order constraints required by nesting. However, while this technique works well at the scale it was evaluated (4 speculative tasks), it would require an impractical number of timestamp bits at the scale we consider (4096 speculative tasks). Moreover, this technique would cause over-serialization and does not support exposing timestamps to programs.

4.6.3 Nesting With Non-speculative Parallelism

Nesting is supported by most parallel programming languages, such as OpenMP [70]. In many languages, such as NESL [26], Cilk [81], and X10 [46], nesting is the natural way to express parallelism. Supporting nested parallelism in these non-speculative systems is easy because parallel tasks have no atomicity requirements: they either operate on disjoint data or use explicit synchronization, such as locks [49] or dataflow annotations [69], to avoid data races. Though nested non-speculative parallelism is often sufficient, many algorithms need speculation to be parallelized efficiently [155]. By making nested speculative parallelism practical, Fractal brings the benefits of composability and fine-grained parallelism to a broader set of programs.

4.7 Summary

We have presented Fractal, a new execution model for fine-grained nested speculative parallelism. Fractal lets programmers compose ordered and unordered algorithms without undue serialization. Our Fractal implementation builds on the Swarm architecture and relies on a dynamically chosen task order to perform fine-grained speculation, operating at the level of individual tasks. Our implementation sidesteps the scalability issues of parallel-nesting HTMs and requires simple hardware. We have shown that Fractal can parallelize a broader range of applications than prior work, and outperforms prior speculative architectures by up to 88×.

5

Amalgam: Matching Speculation Resources to Application Needs

A vital function addressed by systems that support speculative parallelism is how to correctly maintain the data dependences across tasks of an application. This involves tracking the address sets (either reads or writes) of tasks to enable conflict detection. Since real world applications are complex and diverse, any design needs to efficiently handle tasks of varying sizes, and in particular support tracking arbitrarily large address sets.

Hardware signatures are a promising solution to track the read and write sets of a task [42]. A hardware signature supports an *approximate* representation of an unbounded set of addresses with a bounded amount of state. The size of the signature affects the performance—false positives can lead to unnecessary aborts—and impacts the implementation costs (area and energy). We focus on Bloom-filter based hardware signatures, which are a popular solution in many speculative architectures. All prior hardware techniques that support speculative parallelism allocate Bloom filters at the granularity of tasks—one Bloom filter each to track the read and write set of a task. Further, these Bloom filters are conservatively sized at design time to ensure low false positive rates across the spectrum of commonly occurring task sizes, as anticipated by the system designer. This can, however, lead to inefficient utilization for tiny tasks, while also offering no protection against high false positive rates for applications with very large tasks. Ideally, we would like to *match* the Bloom filter resources to the size of the address set being tracked.

The Swarm architecture affords this capability by providing a pool of Bloom filters in hardware as part of the commit queue (Chapter 3). Unlike TM and TLS systems that maintain only one read and one write Bloom filter per core, Swarm's hardware support for queueing provides many more Bloom filters than the number of cores. This provides the architecture choice on how to apportion these Bloom filters to various tasks.

This chapter presents *Amalgam*, a microarchitectural technique that enables matching the Bloom filter resources to the size of the task address sets. We enhance the Swarm hardware architecture with two mechanisms to achieve this. *Signature splitting* breaks

Application	Read Set Size			Write Set Size		
	Median	90th	Max	Median	90th	Max
astar	2	13	13	0	1	2
color	5	10	12164	1	1	451
bfs	4	5	6	0	1	1
mis	1	2	5	0	1	1
sssp	3	7	10	0	1	1
intruder	27	55	152	3	3	27
vacation	62	132	248	3	10	27
nocsim	5	18	1486	1	6	2519
silo	37	43	315	4	12	77
kmeans	9	72	72	1	3	10

Table 5.1: Address set size of tasks (in 64-byte cache lines) of representative applications.

up a large address set into multiple smaller address subsets, tracking each subset using one Bloom filter from the hardware pool. *Signature merging* combines multiple small address sets tracking them together using a single Bloom filter from the hardware pool. Both mechanisms tradeoff the efficiency of tracking the address sets (and associated false positive rates), against the speculation window offered to applications.

Amalgam enables Swarm to respond gracefully to tasks of different sizes. We evaluate Amalgam in simulation, across several design points using different sizes of Bloom filters and different commit queue sizes. At 256 cores, Amalgam using small Bloom filters (512-bit 8-way) achieves 5% higher performance than Swarm with conservatively-sized, larger Bloom filters (2048-bit 8-way). Unlike Swarm, where individual application performance can degrade by up to 14× due to mismatch in Bloom filter resources and task size, Amalgam's performance is congruent to the net available Bloom filter resources on chip. Amalgam can improve performance for a given system configuration, or reduce implementation costs, reducing the commit queue area by up to 4× without impacting performance.

5.1 Motivation

5.1.1 Application Tasks are Diverse

To illustrate the diversity in application tasks, we profile the size of the read and write set of tasks across different applications from Swarm and STAMP benchmark suite (Section 3.4). Table 5.1 summarizes the results. First, tasks across different applications have different median address set sizes, ranging from 1 64-byte cache line in `mis` to 62 64-byte cache lines in `vacation`. While `vacation` might benefit from using large Bloom filters, say 2048-bit 8-way, the same filter is clearly wasteful for `mis`. Second, tasks within an application can also exhibit significant diversity. `mis` and `bfs` have uniformly small address sets. However, `color` and `nocsim` show greater variance. Extremely large footprints, such as in `color`, result in several false conflicts even with 2048-bit Bloom filters. As we shall see in Sec-

tion 5.1.2 even a few tasks with large footprint can cause false conflicts with several other tasks over a long period during the course of execution, severely degrading performance.

5.1.2 Impact of Bloom Filter Size on Performance

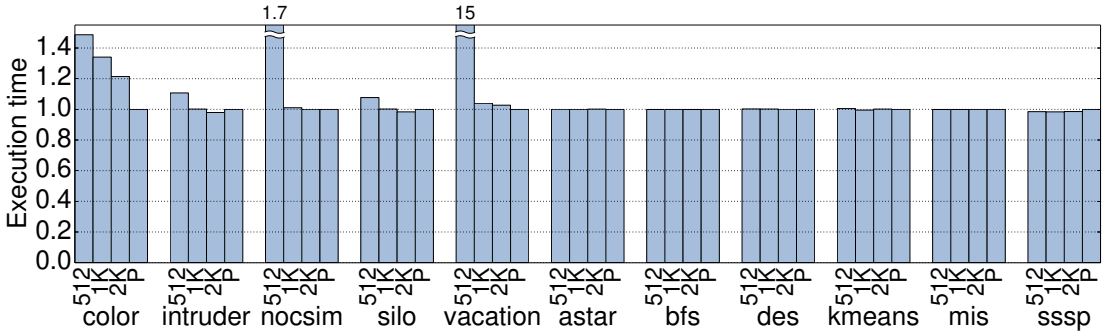


Figure 5-1: Sensitivity of application performance to various Bloom filter sizes (512: 512-bit 8-way, 1K: 1024-bit 8-way, 2K: 2048-bit 8-way) compared to idealized precise address set tracking. Most ordered applications have fine-grained tasks with small footprints and are insensitive to Bloom filter size.

The diversity in tasks, both within and across applications, means that the size of the Bloom filter impacts performance to varying degrees for different applications. Figure 5-1 reports the relative performance of a 256-core Swarm system when using different Bloom filter sizes as well as with an idealized precise tracking of address sets (see Section 5.3 for methodology, modeled system). For example, `astar` has tasks with small address sets, and is accordingly unaffected by the size of the Bloom filter. By contrast, `vacation` has tasks with large address sets, and sees performance drop by 15× when using 512-bit 8-way Bloom filters compared to tracking address sets precisely. The smaller Bloom filters result in larger false positive rates, which translate to more aborts and reduced performance. `color` has relatively small median and 90th percentile address set size. However, the presence of even a few large tasks leads to several false conflicts, resulting in severely degraded performance. With precise tracking, `color` achieves 20% higher performance than when using 2048-bit 8-way Bloom filters, and 45% higher performance than using 512-bit 8-way Bloom filters.

5.1.3 Impact of Commit Queue Size on Performance

The size of the commit queue corresponds to the window of speculation offered to the application. For ordered applications, the window of speculation impacts the parallelism that can be uncovered. Figure 5-2 reports the relative performance of a 256-core Swarm system when using different commit queue sizes. Increasing the commit queue size improves performance for all ordered applications. Many of the ordered applications have fine-grained

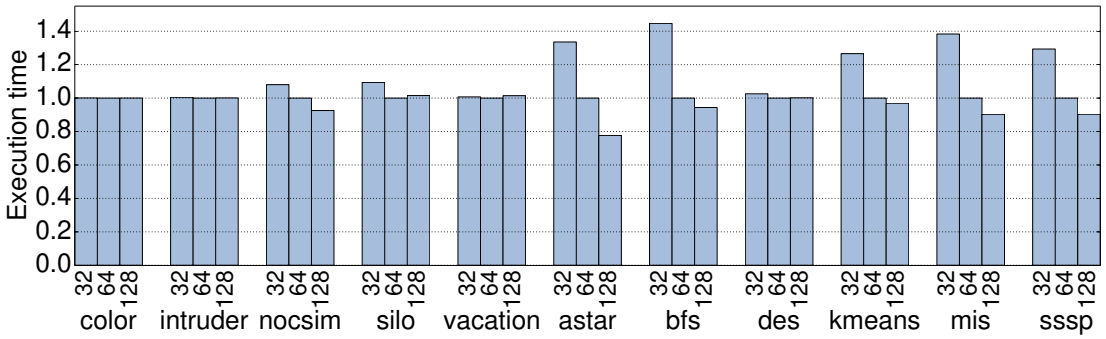


Figure 5-2: Sensitivity of application performance to commit queue size (32, 64, 128 entries per tile) when using idealized precise address set tracking. Unordered applications are insensitive to the commit queue size, while performance of ordered applications improves with larger commit queue size. All numbers are normalized to execution time with 64 commit queue entries per tile.

tasks with small address sets. Using large Bloom filters—which are the dominant contributor to commit queue area—is wasteful for these applications. At the same time, we cannot make the Bloom filter too small even to track tiny address sets—as seen in Section 2.5 small bit-vectors lead to much higher false positive probability. Instead, tracking multiple tiny address sets using the same Bloom filter can lead to more efficient utilization without impacting performance adversely. Unordered applications are largely unimpacted by the commit queue size because the their tasks commit quickly. Tasks either abort on conflict, or commit immediately on completion. However, as seen in Section 5.1.2, these applications are sensitive to Bloom filter size. Thus, leveraging the unused Bloom filters in the commit queue to track large address sets more effectively can reduce the false positive rate and improve performance.

5.2 Amalgam Design

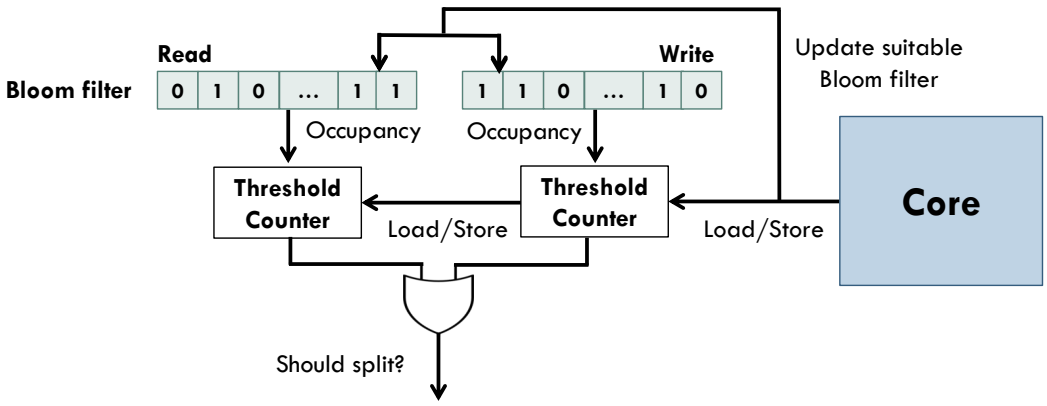
Amalgam builds on the Swarm architecture as described in Chapter 3. Amalgam enhances Swarm to match Bloom filter resources to the address footprint of the tasks. We focus on the Bloom filter since they are the dominant contributor to the speculative state of the task (occupying more than 90% of the commit queue area, see Section 3.3.9), and because they impact the performance of applications (Section 5.1). While Swarm allots each task a single read and write Bloom filter from the pool, in Amalgam, we manage these filters more carefully. We augment the Swarm hardware with two mechanisms: signature splitting and signature merging. We first describe these mechanisms qualitatively, and then analyze the microarchitecture in more detail in Section 5.2.3.

5.2.1 Signature Splitting

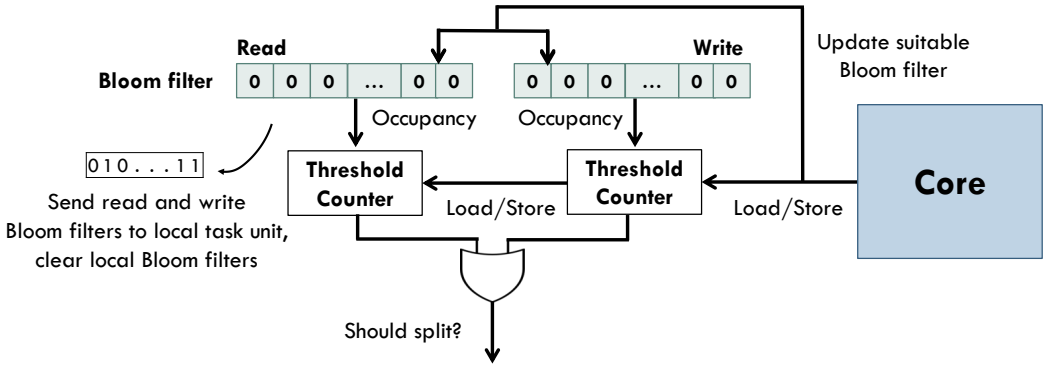
Signature splitting responds to tasks with large address footprints. Large address sets incur higher false positive probability, and consequently lead to more aborts and reduced perfor-

mance. Signature splitting enables tracking a large address set across several Bloom filters. Each Bloom filter tracks a subset of addresses thus keeping the individual false positive probability low. In concert, the Bloom filters track the large address set without affecting the correctness of conflict resolution.

Modifications to the core: To perform signature splitting, we augment every core with threshold counters, one each for the read and write Bloom filter, as shown in Figure 5-3. On every load and store performed by a speculative task, each threshold counter determines if the corresponding address set for this task should be split. If the decision of either threshold counter is to split the address set, the core transfers both the read and write Bloom filters at the core to the local tile's commit queue, and requests a new slot in the commit queue to store the new Bloom filters for the remainder of the task execution. It also clears its local read and write Bloom filters, tracking all future loads and stores by the tasks with fresh Bloom filters.



(a) Threshold counters query occupancy of the Bloom filters.



(b) On a split, transfer Bloom filters to local task unit. Clear local Bloom filters and track addresses using fresh Bloom filters.

Figure 5-3: Signature splitting allows Amalgam to track large address sets across multiple Bloom filters.

Modifications to the task unit: When a core sends a request to split an address set, the local task unit checks if Bloom filters (one each for read and write sets) is available from the pool. If filters are available, then the Bloom filters transferred from the core are stored at the existing Bloom filters associated with the task. The new filters are then associated with the same task. If filters are not available, the task unit simply stores the transferred Bloom filters to the existing Bloom filters associated with the task without allocating any new filters. When it receives the fresh Bloom filters from the core in the future, it simply merges them into the current Bloom filter using a bitwise-OR. During conflict checks, since multiple Bloom filters may be associated with the same task, a potential conflict flagged by any of the Bloom filters, causes all the filters associated with the task to be cleared after the task is aborted. The microarchitecture is described in greater detail in Section 5.2.3.

Threshold counter: We employ a simple threshold counter to determine if an address set should be split. For a given Bloom filter size, we determine the number of elements (T_n) at which the false positive probability exceeds a threshold (T_{FP}). We translate T_n to the *occupancy*—defined as the number of bits set—of the Bloom filter at T_n elements using the formula $T_n = \frac{m}{k} ln(\frac{m}{m-T_s})$, where T_s is the number of bits set to 1 in the $m-$bit $k-$ way Bloom filter. We refer to T_s as the *occupancy threshold*. When the occupancy of the Bloom filter exceeds T_s the counter determines that the address set is too large, and asserts a split request.

5.2.2 Signature Merging

Signature merging handles tasks with small address footprints. Small address sets result in inefficient utilization of Bloom filter resources. For example, for an `astar` task that accesses a single 64-bit address, using a 2048-bit Bloom filter is wasteful. At the same time, we cannot simply size Bloom filters for the smallest task footprint—small Bloom filters incur very high false positive probability which causes significant degradation in performance. Signature merging, instead, merges multiple small address sets, tracking them together using a single Bloom filter. This improves the utilization of the Bloom filter, and effectively increases the speculation window offered to the application.

Modifications to the task unit: We trigger signature merging in a lazy manner. When the task unit has a task available to run, but the commit queue is full (i.e., we have no available Bloom filters in the pool), the task unit attempts to merge two Bloom filters and free one of them for the new task. The candidates for merging are selected as per the merging policy (see below). If no candidates are available for merging, the core stalls as before. If two candidate filters are found, they are merged by a simple bitwise-OR operation. All tasks associated with these filters then point to the merged filter. The other filter is finally cleared and allotted to the new task. If a conflict is flagged on a merged Bloom filter (i.e., multiple tasks point to are tracked by the same Bloom filter), then the virtual times of the tasks are compared to the incoming conflict check request's virtual time

to determine which tasks must be aborted. The microarchitecture is described in greater detail in Section 5.2.3.

Merging policies: We implement a simple greedy policy to select candidate Bloom filters for merging. The *occupancy table* stores the occupancy of different Bloom filters. We walk the occupancy table in order and find the first two Bloom filters whose sum of occupancies is less than *occupancy threshold*, and merge them. This simple policy reaps most of the benefits of signature merging, while keeping hardware overheads modest.

5.2.3 Amalgam Microarchitecture

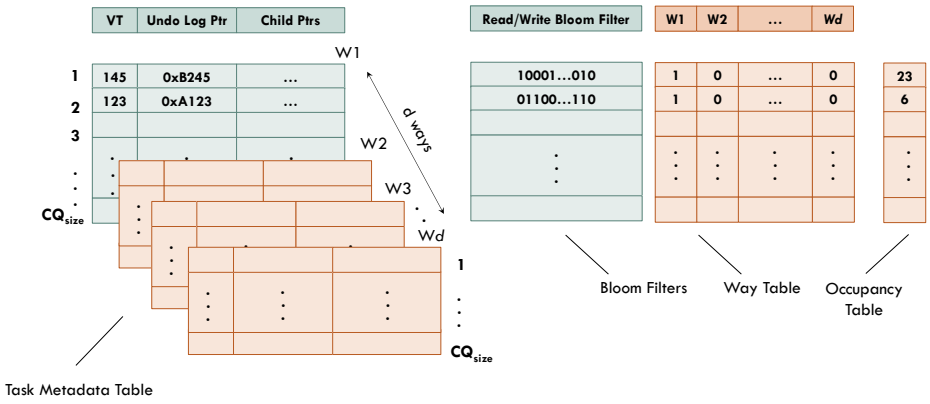
Core: Modifications to the core for signature splitting are simple and require minimal hardware. The shadow Bloom filter incurs a few additional bits per core, the splitting predictor is just a comparator, and we have a simple state machine to handle copying and clearing of the shadow Bloom filter.

Task Unit: Figure 5-4 shows the hardware additions to the task unit, compared to baseline Swarm. Swarm's commit queue comprises two structures that hold the speculative state of tasks that have finished execution but cannot yet commit: (i) task metadata table, which holds the virtual time, undo log pointer, and child pointers, and (ii) Bloom filters to track the read and write address sets. Bloom filters account for more than 90% of the commit queue area, and thus primarily limit the speculation window.

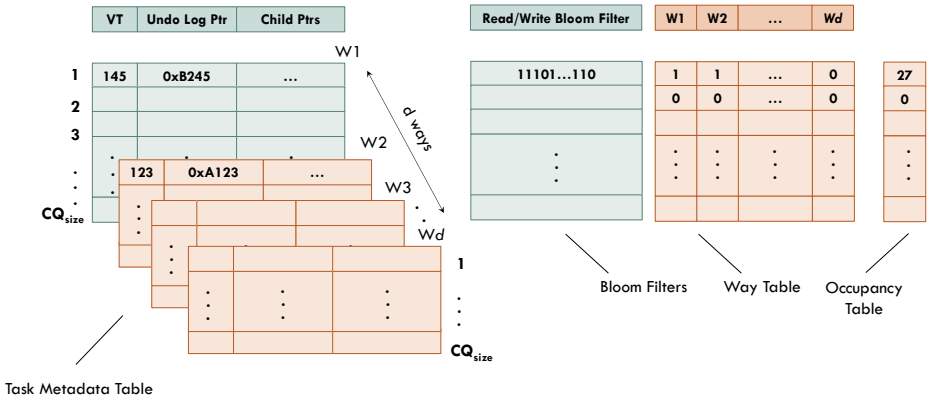
Amalgam's additions are shown in orange. Signature splitting does not require any additional storage. When a task's address set is split, we simply copy over the virtual time and undo log pointer to the entry corresponding to the new Bloom filter in the task metadata table. We re-purpose one of the child pointers to point to the new entry in the task metadata table—this reuses the child task mechanisms to handle task commit and abort correctly.

Signature merging necessitates additional storage. Amalgam adds multiple ways to the task metadata table to track the metadata of tasks associated with merged Bloom filters. The *way table* denotes which ways of a particular entry in the metadata table are valid. We define a *degree of merging* (d), that defines the maximum number of ways, and hence the maximum number of tasks that may be tracked together by a single Bloom filter. The occupancy table tracks the occupancy of each Bloom filter and is used to determine candidate Bloom filters for merging.

Figure 5-4b illustrates the operation on a merge. Say tasks 1 and 2 are merged. The candidate filters are merged by a simple bitwise-OR operation. The metadata of one of the merged tasks, say task 2, is moved to entry 1, way 2. Bits W_1 and W_2 are then set on entry 1 of the way table to indicate this Bloom filter is merged and tracks the address sets of two tasks. If a merged Bloom filter signals a conflict, the virtual times of all the associated tasks are compared and aborts triggered accordingly. This construction allows for easy access to the virtual times during conflict checks, ensuring that conflict checks remain fast. At



(a) State of commit queue structures before a signature merge.



(b) Bloom filters 1 and 2 are merged. Entries in the metadata table are moved and bits in the way table updated accordingly.

Figure 5-4: Modifications to the baseline Swarm task unit for Amalgam. The commit queue comprises a task metadata table, the Bloom filters, a way table, and an occupancy table. Amalgam additions are shown in orange. Figures (a) and (b) show the state of different structures before and after a signature merge.

the same time, the implicit indexing into the additional entries of the metadata table keep hardware overheads low.

We find that a degree of merging (d) of 2 is sufficient to reap most of the gains of signature merging. Compared to a baseline system with 2048-bit Bloom filters and CQ_{size} of 64, this increases the commit queue area by 6%. Halving the CQ_{size} to 32 nearly halves the overall area of the commit queue thus obviating the 6% overhead, while providing the same or better performance compared to the baseline system with $CQ_{size} = 64$.

5.3 Methodology

Modeled system: We use the same methodology as previous chapters (Section 3.4). We model a 256-core Swarm system, as shown in Figure 3-2, with parameters in Table 5.2.

False probability threshold (T_{FP})	0.05%
Degree of merging (d)	2

Table 5.2: Configuration of the 256-core Amalgam system. Core, memory system, and Swarm parameters as listed in Table 3.3.

Benchmarks: We use 9 Swarm applications and 8 applications from the STAMP benchmark suite to evaluate Amalgam. Amalgam impacts the performance of 10 applications (listed in Table 5.1), and hence we evaluate those in detail. For the remaining applications, performance is unaffected by Amalgam.

5.4 Evaluation

We first evaluate the performance of Amalgam against a baseline Swarm system at different design points, and then analyze its behavior in depth.

5.4.1 Amalgam vs. Swarm

Figure 5-5 shows the gmean execution time across the benchmark applications for different design points on both the Swarm architecture and with Amalgam. We report the geometric mean (gmean) execution time for the 10 applications where Amalgam affects performance. All design points are evaluated on a 256-core system with parameters as in Table 5.2, but varying the commit queue size and the Bloom filter size. We evaluate two commit queue sizes (64 and 32 entries per tile), and three Bloom filter configurations (512-bit 8-way, 1024-bit 8-way and 2048-bit 8-way), as well as an idealized precise address set tracking scheme. We label the design points as *CommitQueueSize-BloomFilterSize*. All performance numbers are normalized to the baseline parameters in Table 5.2 (i.e., with 64 commit queue entries per tile and 2048-bit 8-way Bloom filters).

Large Bloom filters (2048-bit or 1024-bit) result in few false conflicts. However, at 512-bits the false positive probability increases for several unordered applications and is the dominant cause for increased gmean execution time. *64-512* is 43% slower than *64-2K*, while *32-512* is 47% slower than *32-2K*. Signature splitting successfully overcomes this problem—the Amalgam variants with 512-bit Bloom filter match the gmean performance using larger Bloom filters.

Smaller commit queue sizes decrease the speculation window, and reduce performance for most ordered applications. Thus, *32-2K* is 17% slower than *64-2K*, while *32-512* is 20% slower than *64-512*. Signature merging reclaims most of this performance loss.

Overall, Amalgam handles tasks of varying sizes gracefully regardless of Bloom filter or commit queue sizes. Amalgam can help improve performance at the same area, or match the performance while offering area savings. *32-1K* with Amalgam matches the performance of *64-2K* at 1/4th the area for Bloom filter resources (saving $0.228mm^2$, 24

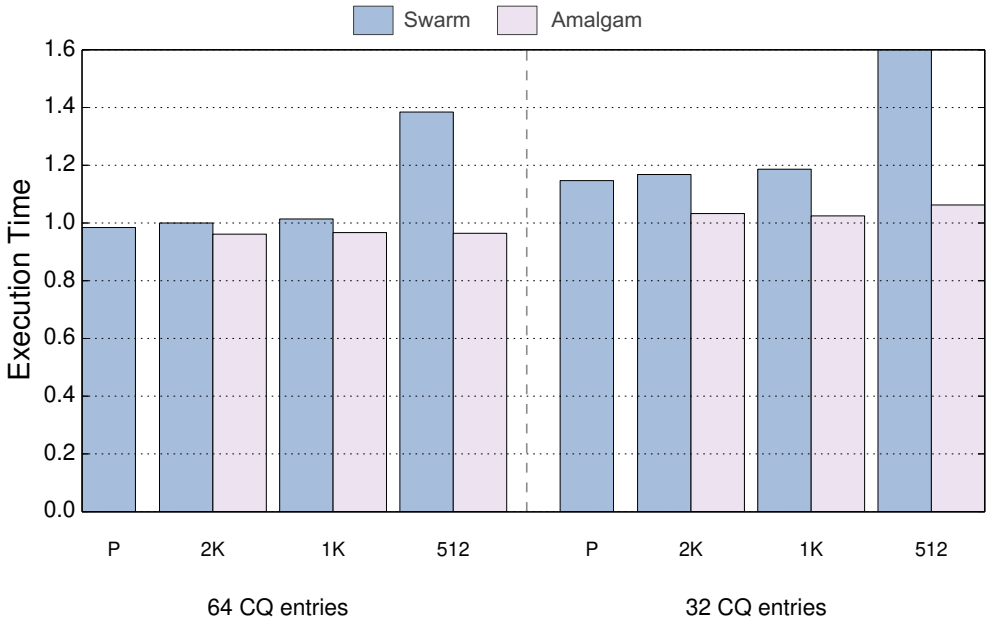


Figure 5-5: Gmean execution time of Swarm and Amalgam at 256-cores for various design points—64, 32 commit queue entries per tile, and 512-bit 8-way (512), 1024-bit 8-way (1K) and 2048-bit 8-way (2K) Bloom filters. P represents an idealized precise address set tracking scheme. Amalgam gracefully responds to tasks of varying sizes regardless of Bloom filter size and number of commit queue entries.

KB of 2-port SRAM per tile), while *32-512* achieves within 6% of the performance of *64-2K* at 1/8th the area for Bloom filter resources (saving $0.266mm^2$, 28 KB of 2-port SRAM area per tile).

5.4.2 Amalgam Analysis

We analyze the behavior of different benchmarks in greater detail to understand the differences in performance of Amalgam compared to Swarm. In Figure 5-6, the height of each bar is the sum of cycles spent by all cores, normalized to the cycles spent with 64 commit queue entries and 2048-bit 8-way Bloom filters (*64-2K*). Each bar shows the percentage of cycles that cores spend on different activities: (i) running tasks that are ultimately committed, (ii) running tasks that are later aborted, (iii) stalled on a full task or commit queue, or (iv) other (queue spills or stalled due to lack of tasks). Each group of bars shows results for a different application at 256 cores. We analyze only two Bloom filter configurations here for simplicity: 2048-bit 8-way and 512-bit 8-way. We evaluate both Swarm and Amalgam with 32 and 64 commit queue entries for both Bloom filter configurations. For simplicity, we show the cycle breakdown for four representative applications. Other applications with qualitatively similar behavior are identified in the text.

`vacation` has tasks with large address sets. This leads to high false positive rates when tracked with a 512-bit 8-way Bloom filter. Compared to 2048-bit Bloom filters,

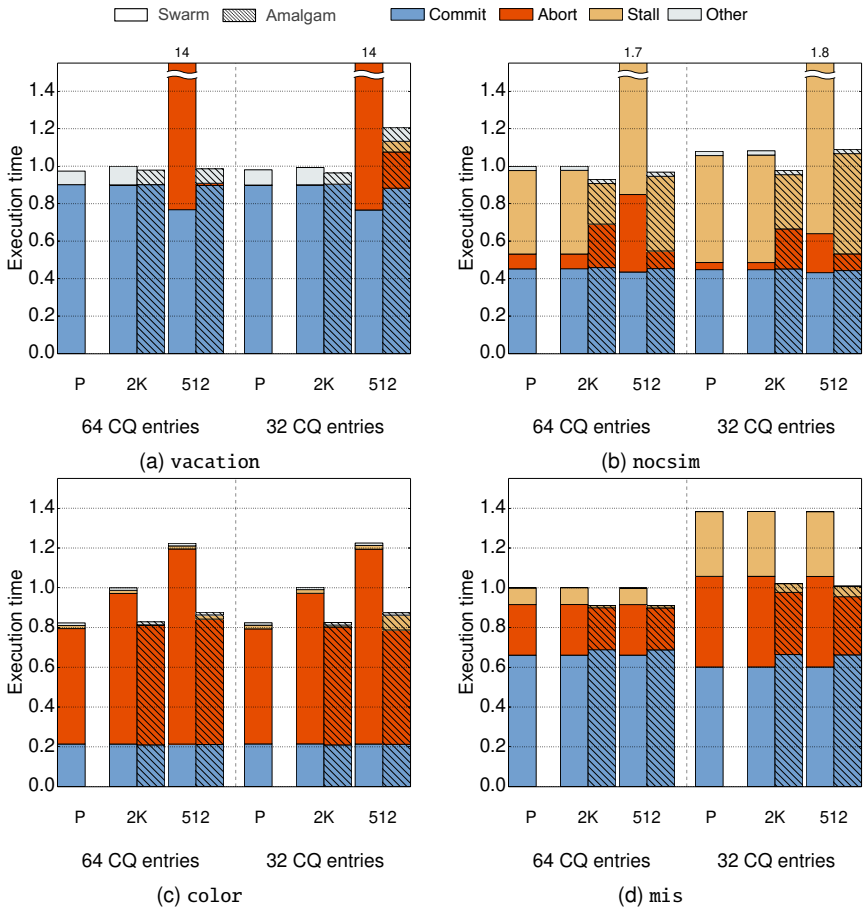


Figure 5-6: Breakdown of core cycles at 256-cores for four representative applications. All numbers are normalized to the 2048-bit 8-way Bloom filter with 64 commit queue entries. Amalgam reduces aborts due to false conflicts when application tasks have large address sets by performing signature splitting ((a), (b) and (c)). Amalgam reduces commit queue stalls by merging tasks with tiny address sets ((b) and (d)).

512-bit Bloom filters show significantly higher aborts and accordingly lower performance (Figure 5-6a). Amalgam overcomes this through signature splitting, reducing the abort rate and improving performance. At 64 commit queue entries, Amalgam performs slightly better than with 2048-bit Bloom filters, and improves performance over Swarm by 14×. However, at 32 commit queue entries, signature splitting introduces a few commit queue stalls as multiple Bloom filters are used to track large tasks. However, the gains from the reduced false conflict rate are more significant, effectively improving performance by 12× compared to Swarm. Other unordered applications from the STAMP benchmark suite such as `intruder` exhibit similar behavior as `vacation`.

Ordered applications `color` and `nocsim` (Figure 5-6c, Figure 5-6b) also benefit from signature splitting on some of their larger tasks. Similar to `vacation`, signature splitting reduces aborts arising from false conflicts for `color`. This improves performance by 40%

and 20% compared to Swarm at 512-bit Bloom filters and 2048-bit Bloom filters respectively, for both commit queue configurations. `nocsim`, however, benefits from both signature splitting and signature merging, with the latter reducing commit queue stalls. This improves performance by 70% over Swarm at 512-bit Bloom filters for both commit queue configurations. The remaining ordered applications mostly have fine-grained tasks that do not require signature splitting. Most of the benefits for these applications arise from signature merging providing a larger speculation window, and reducing commit queue stalls. Signature merging has a greater impact when the hardware has smaller number of commit queue entries, which places more pressure on the commit queue. Thus, `mis` sees a 39% improvement in performance compared to Swarm with 32 commit queue entries, almost matching the performance with 64 commit queue entries at 512-bit Bloom filters. Other ordered applications, `astar`, `bfs` and `sssp` also benefit from signature merging in a similar manner.

5.4.3 Sensitivity Studies

We explore Amalgam's sensitivity to different parameters at 256 cores.

Degree of merging: We sweep the degree of merging (d) from $d = 2$ to $d = 10$. With the greedy merge policy, the gmean speedup does not vary by more than 2% across different different degrees of merging. We use $d = 2$ since it reaps all the benefits while incurring minimal hardware overhead in the task metadata table and index table.

Occupancy threshold: We explore different choices for the false positive probability T_{FP} and correspondingly the occupancy threshold for splitting and merging T_s. We find that $T_{FP} < 0.05\%$ sets a suitable threshold for avoiding false conflicts for different sizes of Bloom filters. Lower values of T_{FP} increase the number of splits for applications with large tasks. With the exception of `vacation` performance of most applications remains unchanged even at lower T_{FP}. `vacation` suffers a 10% reduction in performance due to greater splitting resulting in more commit queue stalls. Applications that benefit from merging have fine-grained tasks and are unaffected by lower values of T_{FP}.

5.5 Summary

We have presented Amalgam, a microarchitectural technique that enables matching Bloom filter resources for speculation to task size. Amalgam enables Swarm to respond gracefully to tasks of different sizes. By efficiently utilizing available Bloom filter resources on chip, Amalgam can improve performance for a given system configuration, or reduce implementation costs without impacting performance.

6

Conclusions and Future Work

To conclude this dissertation, we summarize the contributions of this dissertation and outline areas for future work.

6.1 Contributions

Current multicores are limited in the range of parallelism they can exploit in applications. In particular, they cannot harness fine-grained parallelism that is crucial to scaling many applications and enabling widespread parallelism. This is because current systems lack efficient support for fine-grained tasks and synchronization. Our key insight is that *order* is a simple, general and powerful synchronization primitive. We have shown that architectural support for order unlocks abundant fine-grained parallelism in applications. Specifically we have made the following contributions:

- *Swarm*, an architecture that successfully mines ordered parallelism, which is abundant, but difficult to exploit using current software and hardware techniques. Swarm relies on a co-designed execution model and microarchitecture to scale efficiently. Swarm executes tasks speculatively and out of order, and efficiently speculates thousands of tasks ahead of the earliest active task to uncover enough parallelism. Swarm adapts existing eager version management and conflict detection schemes, and contributes several new techniques that allow it to scale to large core counts and speculation windows.

- *Fractal*, a new execution model for nested speculative parallelism that supports both unordered and ordered speculation. Fractal decouples atomicity from parallelism by introducing the notion of a domain. Fractal programs consist of tasks located in a hierarchy of nested domains. Any task can create a new subdomain, and all tasks in a domain appear to execute atomically with respect to tasks outside the domain.

Our Fractal implementation builds on the Swarm architecture and supports arbitrary nesting levels cheaply. Unlike prior work, our implementation focuses on extracting parallelism at the finest (deepest) levels first. It guarantees the atomicity of large domains by enforcing an order among individual tasks rather than building impractically large atomic units. This allows the system to manage tiny speculative tasks that are easy to track in hardware, reaping the benefits of fine-grained parallelism, and avoids the problems that plague prior nested-parallel HTMs.

- *Amalgam*, a microarchitectural technique that enables matching the speculation resources allotted to a task to the task size. Signature splitting tracks large address sets cooperatively using multiple Bloom filters, and signature merging tracks multiple small address sets using a single Bloom filter. Amalgam gracefully handles diverse tasks across applications and improves the efficiency of speculative execution reducing implementation costs.

Our results demonstrate significant improvements in performance for applications from a broad set of domains, some of which have eluded successful parallelization for decades. Besides achieving near-linear scalability, the resulting programs are almost as simple as their sequential counterparts, as they do not use explicit synchronization.

6.2 Future Work

The techniques in this dissertation open a number of avenues for future work, described below:

- *Eliminating the need for global cache coherence:* Cache coherence is a staple of modern multicores. While coherence protocols can scale to thousands of cores, their added complexity and overhead make them undesirable at very large scales, e.g., in a machine spanning multiple boards or racks. Accurate spatial hints can make global coherence unnecessary: if all accesses to shared read-write data are localized within a tile, the coherence protocol becomes superfluous across tiles. We have shown that hints effectively localize the vast majority of read-write accesses with fine-grained tasks. Further, Fractal allows a task to be divided at any arbitrary point. Thus, if a task needs data from a different tile, its context can be migrated automatically while preserving atomicity and without requiring global coherence.

- *Towards a practical auto-parallelizing compiler:* Automatic parallelization has proven daunting, with limited success despite decades of work. Without support for speculative execution, parallelizing compilers are limited by the precision of dependence analysis, and cannot effectively decompose even simple algorithms into parallel tasks. But even with speculative execution, prior parallelizing compilers have attained lim-

ited speedups. A key part of the problem is that the speculative execution models targeted by these compilers (e.g., TLS) do not support composition. Therefore, prior TLS compilers have focused on parallelizing a single control structure (e.g., a `for` loop) at a time. This is insufficient in large programs, where parallelism often arises across function and compilation unit boundaries. Fractal's support for unrestricted nesting enables compilers to exploit parallelism across multiple control regions in a clean fashion, while maintaining their atomicity and order requirements. This could lay the groundwork towards realizing a practical auto-parallelizing compiler.

- *Silicon implementations and deeper understanding of components:* This dissertation has studied and developed the broad architectural techniques for exploiting ordered and nested speculative parallelism. However, a deeper understanding is required for each of the components developed in this thesis, including a comprehensive exploration of the design space (e.g., size of the task queue, commit queue, number of cores per tile etc.), an extensive evaluation of various policies (e.g., admission, eviction and spilling policies for the task and commit queue, etc.), and a thorough understanding of the cost-performance tradeoffs to guide an actual implementation. Real silicon implementations can not only crystallize these efforts, but also provide confidence on the practicality of the ideas developed in this dissertation. A first step could be an FPGA implementation to serve as a vehicle to validate the correctness of the architectural techniques we have proposed. This effort could also lead to the development of stand-alone accelerators for specific applications (such as discrete-event simulation or graph analytics).

- *Development of tools and theory to analyze and understand speculation performance:* Programmers can benefit from tools that identify and help diagnose scalability bottlenecks and performance pathologies. Speculative execution demands new tools and techniques, for example to determine cause for aborts, identify resource constraints (such as task and commit queue resources), creation and visualization of parallelism profiles, and more. The development of a more rigorous theory on speculative execution could not only aid our understanding of the benefits of speculation, but importantly identify the limitations of speculative execution. This can also guide the design of performance models and offer stronger guarantees on the performance of speculation (e.g., given the ideal parallelism in an application, can we guarantee under specific circumstances that speculative execution will deliver within a constant factor of the ideal parallelism?). The development of such tools and theories would encourage adoption of the techniques developed in this dissertation by programmers and computer architects.

- *Hardware and software co-design:* We believe the design process in this dissertation could serve as a template for developing future parallel architectures. We adopt a vertically-integrated approach of studying the software and hardware architecture

in concert to understand the scalability challenges of applications. We also co-design the execution model and architectural support sidestepping the limitations of prior hardware- and software-only approaches. We believe that this design approach generalizes beyond this dissertation: many inefficiencies in current parallel systems are caused by a *semantic gap* between hardware and software, and innovating across the hardware-software interface is an effective way to bridge this gap. New interfaces could allow applications to convey which tasks are resilient to conflicts to reduce aborts (e.g., in iterative methods where conflicting updates cause small, temporary errors that do not jeopardize convergence), or capture application semantics such as commutativity to avoid aborts, or to express what tasks are safe to execute non-speculatively. These could also be coupled with other techniques such as approximate computing where strict accuracy is not essential. Importantly, such an approach compels system designers to carefully consider the problem to be solved, the characteristics of the range of algorithms to solve the problem, and the requisite software and hardware support to realize the best performance.

THIS dissertation has laid a solid foundation for exploiting fine-grained speculative parallelism. We hope the techniques we have developed provide guidance to computer architects in designing versatile parallel systems. More broadly, we hope this work paves the path towards a world where parallelism is pervasive and parallel programming is the default.

Bibliography

[1] "9th DIMACS Implementation Challenge: Shortest Paths," 2006.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proc. SPAA*, 2000.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Proc. ISPASS*, 2009.

[4] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of X10 computations with bounded resources," in *Proc. SPAA*, 2007.

[5] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *Proc. PPoPP*, 2008.

[6] K. Agrawal, C. E. Leiserson, and J. Sukha, "Memory Models for Open-nested Transactions," in *Proc. MSPC*, 2006.

[7] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A scalable relaxed priority queue," in *Proc. PPoPP*, 2015.

[8] G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development*, Apr. 1964.

[9] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS*, 1967.

[10] *4096x128 ternary CAM datasheet (28nm)*, Analog Bits, 2011.

[11] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. HPCA-11*, 2005.

[12] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, 1994.

[13] R. J. Anderson and J. C. Setubal, "On the parallel implementation of Goldberg's maximum flow algorithm," in *Proc. SPAA*, 1992.

[14] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *Proc. HiPEAC*, 2009.

[15] A. Armejach, A. Negi, A. Cristal, O. Unsal, P. Stenstrom, and T. Harris, "HARP: Adaptive abort recurrence prediction for hardware transactional memory," in *HiPC-20*, 2013.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[17] T. Austin and G. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proc. ISCA-19*, 1992.

[18] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *10th DIMACS Implementation Challenge Workshop*, 2012.

[19] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Implementing and evaluating nested parallel transactions in software transactional memory," in *Proc. SPAA*, 2010.

[20] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Making nested parallel transactions practical using lightweight hardware support," in *Proc. ICS'10*, 2010.

[21] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, "The OpenTM transactional application programming interface," in *Proc. PACT-16*, 2007.

[22] N. Baumstark, G. Blelloch, and J. Shun, "Efficient implementation of a synchronous parallel push-relabel algorithm," in *Proc. ESA*, 2015.

[23] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *Proc. MICRO-42*, 2009.

[24] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom filter guided transaction scheduling," in *Proc. MICRO-44*, 2011.

[25] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proc. PPoPP*, 2012.

[26] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proc. PPoPP*, 1993.

[27] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee, "Implementation of a portable nested data-parallel language," *Journal of parallel and distributed computing*, 21(1), 1994.

[28] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, 13(7), 1970.

[29] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, 46(5), 1999.

[30] C. Blundell, J. Devietti, E. C. Lewis, and M. M. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proc. ISCA-34*, 2007.

[31] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "TokenTM: Efficient execution of large transactions with hardware transactional memory," in *Proc. ISCA-35*, 2008.

[32] 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 *Proc. ISCA*, 2007.

[33] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *Proc. PPoPP*, 2008.

[34] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A Generic Distributed DAG Engine for High Performance Computing," *Parallel Computing*, 38, 2012.

[35] S. Brand and R. Bidarra, "Multi-core scalable and efficient pathfinding with Parallel Ripple Search," *Computer Animation and Virtual Worlds*, 23(2), 2012.

[36] S. E. Breach, "Design and Evaluation of a Multiscalar Processor," Ph.D. dissertation, 1998, aAI9910432.

[37] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A Practical Concurrent Binary Search Tree," in *Proc. PPoPP*, 2010.

[38] W. Buchholz, *Planning a Computer System: Project Stretch*. New York, NY, USA: McGraw-Hill, Inc., 1962.

[39] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proc. ISCA-18*, 1991.

[40] J. L. Carter and M. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. STOC-9*, 1977.

[41] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency," in *Proc. ISCA-34*, 2007.

[42] L. Ceze, J. Tuck, J. Torrellas, and C. Caşcaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proc. ISCA-33*, 2006.

[43] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SDM*, 2004.

[44] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "ZPL: A Machine Independent Programming Language for Parallel Computers," *IEEE Transactions on Software Engineering*, Mar. 2000.

[45] S. Chandrasekaran and M. D. Hill, "Optimistic simulation of parallel architectures using program executables," in *PADS*, 1996.

[46] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. OOPSLA-20*, 2005.

[47] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of Pointers and Structures," in *Proc. PLDI*, 1990.

[48] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on cmps," in *Proc. SPAA*, 2007.

[49] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark, "Detecting data races in Cilk programs that use locks," in *Proc. SPAA*, 1998.

[50] B. V. Cherkassky and A. V. Goldberg, "On implementing the push-relabel method for the maximum flow problem," *Algorithmica*, 19(4), 1997.

[51] J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects," in *Proc. POPL*, 1993.

[52] K. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a Bloom filter," *Information Processing Letters*, 110(21):944–949, 2010.

[53] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-based transactional memory," in *Proc. ASPLOS-XII*, 2006.

[54] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "Tradeoffs in transactional memory virtualization," in *Proc. ASPLOS-XII*, 2006.

[55] M. Cintra and D. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *PPoPP*, 2003.

[56] M. Cintra, J. F. Martínez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-memory Multiprocessors," in *Proc. ISCA-27*, 2000.

[57] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, , and D. Woodford, "Spanner: Google's globally distributed database," *ACM TOCS*, 31(3), 2013.

[58] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[59] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *VLDB*, 2010.

[60] M. Das, "Unification-based Pointer Analysis with Directional Assignments," in *Proc. PLDI*, 2000.

[61] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, 38(1), 2011.

[62] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A clustered manycore processor architecture for embedded and accelerated applications," in *Proc. HPEC*, 2013.

[63] A. Deutsch, "Interprocedural May-alias Analysis for Pointers: Beyond K-limiting," in *Proc. PLDI*, 1994.

[64] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared memory multiprocessing," in *Proc. ASPLOS-XIV*, 2009.

[65] N. Diegues and J. Cachopo, "Practical parallel nesting for software transactional memory," in *Proc. DISC*, 2013.

[66] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic Scheduling for Hardware Transactional Memory," in *Proc. SPAA*, 2015.

[67] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *Proc. PODC-27*, 2008.

[68] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *Proc. PODC-28*, 2009.

[69] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, 21(02), 2011.

[70] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *IWOMP-4*, 2008.

[71] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye, "Optimizations and oracle parallelism with dynamic translation," in *Proc. MICRO-32*, 1999.

[72] M. Ekman, F. Warg, and J. Nilsson, "An In-depth Look at Computer Performance Growth, 2005.

[73] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," in *Proc. PLDI*, 1994.

[74] E. Fatehi and P. Gratz, "ILP and TLP in shared memory applications: a limit study," in *Proc. PACT-23*, 2014.

[75] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, 1987.

[76] A. Ferscha and S. Tripathi, "Parallel and distributed simulation of discrete event systems," U. Maryland, Tech. Rep., 1998.

[77] A. Ferscha and S. K. Tripathi, "Parallel and Distributed Simulation of Discrete Event Systems," Tech. Rep., 1994.

[78] R. J. Figueiredo and J. A. B. Fortes, "Hardware Support for Extracting Coarse-Grain Speculative Parallelism in Distributed Shared-Memory Multiprocesors," in *Proc. ICPP*, 2001.

[79] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," *IEEE Trans. Comput.*, 1996.

[80] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *FOCS*, 1984.

[81] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. PLDI*, 1998.

[82] S. L. Fung and J. G. Steffan, "Improving cache locality for thread-level speculation," in *Proc. IPDPS*, 2006.

[83] S. Garold, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, 19(10), 1970.

[84] 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 *Proc. HPCA-9*, 2003.

[85] M. J. Garzarán, M. Prvulovic, V. Viñals, J. M. Llabería, L. Rauchwerger, and J. Torrellas, "Using Software Logging to Support Multi-Version Buffering in Thread-Level Speculation," in *Proc. PACT-12*, 2003.

[86] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc http://goog-perftools.sourceforge.net/doc/tcmalloc.html," 2007.

[87] R. Ghiya and L. J. Hendren, "Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C," in *Proc. POPL*, 1996.

[88] D. Goldfarb and M. D. Grigoriadis, "A computational comparison of the dinic and network simplex methods for maximum flow," *Annals of Operations Research*, 13(1), 1988.

[89] G. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," *IBM Journal of Research and Development*, Jan. 1990.

[90] J. P. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles, C. Young, and D. E. Shaw, "Hardware support for fine-grained event-driven computation in Anton 2," in *Proc. ASPLOS-XVIII*, 2013.

[91] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *Proc. IPDPS*, 2010.

[92] B. Hackett and R. Rugina, "Region-based Shape Analysis with Tracked Locations," in *Proc. POPL*, 2005.

[93] D. C. Halbert and P. B. Kessler, "Windows of overlapping register frames," CS 292R Final Report, UC Berkeley, 1980.

[94] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *Proc. ASPLOS-VIII*, 1998.

[95] 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 *Proc. ISCA-31*, 2004.

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

[97] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. on Systems Science and Cybernetics*, 4(2), 1968.

[98] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proc. SPAA*, 2014.

[99] M. A. Hassaan, D. Nguyen, and K. Pingali, "Brief announcement: Parallelization of asynchronous variational integrators for shared memory architectures," in *Proc. SPAA*, 2014.

[100] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *Proc. PPoPP*, 2011.

[101] M. A. Hassaan, D. Nguyen, and K. Pingali, "Kinetic Dependence Graphs," in *Proc. ASPLOS-XX*, 2015.

[102] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit, "A Lazy Concurrent List-based Set Algorithm," in *Proc. OPODIS*, 2005.

[103] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach (6th ed.)*. Morgan Kaufmann, 2017.

[104] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. ISCA*, 1993.

[105] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[106] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Distributed Computing*. Springer, 2005, pp. 324–338.

[107] B. Holt, P. Briggs, L. Ceze, and M. Oskin, "Alembic: automatic locality extraction via migration," in *Proc. OOPSLA*, 2014.

[108] D. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas, "Two hardware-based approaches for deterministic multiprocessor replay," *Comm. ACM*, 2009.

[109] Intel Corporation, "Intel 64 and IA-32 architectures software developerâĂŹs manual."

[110] T. Issariyakul and E. Hossain, *Introduction to network simulator NS2*. Springer, 2011.

[111] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar, "Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies," in *Proc. ASPLOS-XVIII*, 2013.

[112] D. R. Jefferson, "Virtual time," *ACM TOPLAS*, 7(3), 1985.

[113] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *Proc. MICRO-49*, 2016.

[114] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *Proc. MICRO-48*, 2015.

[115] J. Jun, S. Jacobson, J. Swisher *et al.*, "Application of discrete-event simulation in health care clinics: A survey," *Journal of the operational research society*, 50(2), 1999.

[116] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," in *Proc. ISCA-25*, 1998.

[117] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proc. ASPLOS-X*, 2002.

[118] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Scheduling strategies for optimistic parallel execution of irregular programs," in *Proc. SPAA*, 2008.

[119] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," in *Proc. ASPLOS-XIII*, 2008.

[120] S. Kumar, C. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. ISCA-34*, 2007.

[121] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, 6(2), 1981.

[122] M. Lam and R. Wilson, "Limits of control flow on parallelism," in *Proc. ISCA-19*, 1992.

[123] J. R. Larus and P. N. Hilfinger, "Detecting Conflicts Between Structure Accesses," in *Proc. PLDI*, 1988.

[124] C. Leiserson and T. Schardl, "A work-efficient parallel breadth-first search algorithm," in *Proc. SPAA*, 2010.

[125] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 209–221.

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

[127] A. Lew, J. Marsden, M. Ortiz, and M. West, "Asynchronous variational integrators," *Arch. Rational Mech. Anal.*, 167(2), 2003.

[128] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proc. OPODIS-XVII*, 2013.

[129] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, 2008.

[130] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Proc. SOSP-23*, 2011.

[131] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.

[132] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, 48(2), October 2015.

[133] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," in *Proc. ISCA-33*, 2006.

[134] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proc. IISWC*, 2008.

[135] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proc. ISCA-34*, 2007.

[136] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys (CSUR)*, 18(1), 1986.

[137] K. Moore, J. Bobba, M. Moravan, M. D. Hill, and D. Wood, "LogTM: Log-based transactional memory," in *Proc. HPCA-12*, 2006.

[138] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, "Supporting nested transactional memory in LogTM," in *Proc. ASPLOS-XII*, 2006.

[139] J. E. B. Moss, "Open Nested Transactions: Semantics and Support, 2006.

[140] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and architecture sketches," *Science of Computer Programming*, 63(2), 2006.

[141] J. E. Nelson, "Latency-Tolerant Distributed Shared Memory For Data-Intensive Applications," Ph.D. dissertation, 2015.

[142] D. Nguyen, A. Lenharth, and K. Pingali, "A Lightweight Infrastructure for Graph Analytics," in *Proc. SOSP-24*, 2013.

[143] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *Proc. ASPLOS-XVI*, 2011.

[144] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, "Open nesting in software transactional memory," in *Proc. PPoPP*, 2007.

[145] A. Nicolau and J. Fisher, "Using an oracle to measure potential parallelism in single instruction stream programs," in *Proc. MICRO-14*, 1981.

[146] K. Nii, T. Amano, N. Watanabe, M. Yamawaki, K. Yoshinaga, M. Wada, and I. Hayashi, "A 28nm 400MHz 4-Parallel 1.6Gsearch/s 80Mb Ternary CAM," in *Proc. ISSCC*, 2014.

[147] K. Nikas, N. Anastopoulos, G. Goumas, and N. Koziris, "Employing transactional memory and helper threads to speedup Dijkstra's algorithm," in *Proc. ICPP*, 2009.

[148] M. Noakes, D. Wallach, and W. Dally, "The J-Machine multicomputer: an architectural evaluation," in *Proc. ISCA-20*, 1993.

[149] NVidia, "Tesla V100 Datasheet," 2018.

[150] OpenMP Application Program Interface, "Version 2.5," 2005.

[151] OpenStreetMap, "http://www.openstreetmap.org."

[152] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, "Controlling program execution through binary instrumentation," *SIGARCH Comput. Archit. News*, 33(5), 2005.

[153] R. Panigrahy and S. Sharma, "Sorting and searching using ternary CAMs," *IEEE Micro*, 23(1), 2003.

[154] A. Pavlo, E. P. Jones, and S. Zdonik, "On predictive modeling for optimizing transaction execution in parallel OLTP systems," *Proc. of the VLDB Endowment*, 5(2), 2011.

[155] 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 *Proc. PLDI*, 2011.

[156] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime compilation techniques for data partitioning and communication schedule reuse," in *Proc. SC93*, 1993.

[157] L. Porter, B. Choi, and D. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *Proc. PACT-18*, 2009.

[158] M. Postiff, D. Greene, G. Tyson, and T. Mudge, "The limits of instruction level parallelism in SPEC95 applications," *Comp. Arch. News*, 27(1), 1999.

[159] M. K. Prabhu, "Parallel Programming Using Thread-level Speculation," Ph.D. dissertation, Stanford, CA, USA, 2006, aAI3197497.

[160] D. Prountzos, R. Manevich, and K. Pingali, "Synthesizing parallel graph programs via automated planning," in *Proc. PLDI*, 2015.

[161] X. Qian, B. Sahelices, and J. Torrellas, "OmniOrder: Directory-based conflict serialization of transactions," in *Proc. ISCA-41*, 2014.

[162] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *Proc. of the 34th annual ACM/IEEE intl. symp. on Microarchitecture*, 2001.

[163] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proc. ISCA-32*, 2005.

[164] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: Transactional Memory for an Operating System," *IEEE Micro*, 28(1), 2008.

[165] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative Parallelization Using Software Multi-threaded Transactions," in *Proc. ASPLOS-XV*, 2010.

[166] K. R. Rao, D. N. Kim, and J.-J. Hwang, *Fast Fourier Transform - Algorithms and Applications*. Springer Publishing Company, Incorporated, 2010.

[167] N. Rapolu, K. Kambatla, S. Jagannathan, and A. Grama, "TransMR: Data-centric programming beyond data parallelism," in *HotCloud*, 2011.

[168] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[169] S. Reinhardt, M. D. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood, "The Wisconsin Wind Tunnel: virtual prototyping of parallel computers," in *SIGMETRICS*, 1993.

[170] 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 *Proc. ICS'05*, 2005.

[171] 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 *Proc. ICS'05*, 2005.

[172] S. Robinson, *Simulation: The Practice of Model Development and Use*. USA: John Wiley & Sons, Inc., 2004.

[173] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proc. PPoPP*, 2010.

[174] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling Stencils in High Performance Fortran," in *Proc. SC97*, 1997.

[175] R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM*, 1978.

[176] M. M. Saad and B. Ravindran, "Hyflow: A high performance distributed software transactional memory framework," in *Proc. HPDC*, 2011.

[177] D. Sainz and H. Attiya, "Relstm: A proactive transactional memory scheduler," in *International Workshop on Transactional Computing (TRANSACT)*, 2013.

[178] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.

[179] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic Fine-Grain Scheduling of Pipeline Parallelism," in *Proc. PACT-20*, 2011.

[180] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing signatures for transactional memory," in *Proc. MICRO-40*, 2007.

[181] D. Sanchez, R. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proc. ASPLOS-XV*, 2010.

[182] M. Schlansker, T. M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson, "Compilers for instruction-level parallelism," *IEEE Computer*, 1997.

[183] S. Schneider, C. Antonopoulos, and D. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *Proc. ISMM-5*, 2006.

[184] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in *Proc. ISCA-35*, 2008.

[185] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proc. PPoPP*, 2013.

[186] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proc. SPAA*, 2012.

[187] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," in *Proc. SPAA*, 2014.

[188] T. Skare and C. Kozyrakis, "Early release: Friend or foe?" in *Proc. WTW*, 2006.

[189] J. E. Smith, G. E. Dermer, B. D. Vanderwarn, S. D. Klinger, and C. M. Rozewski, "The ZS-1 Central Processor," in *Proc. ASPLOS-II*, 1987.

[190] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. of the IEEE*, 1995.

[191] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. ISCA-22*, 1995.

[192] L. Soule and A. Gupta, "An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation," *ACM TOMACS*, 1(4), 1991.

[193] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *Proc. POPL*, 1996.

[194] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede Approach to Thread-level Speculation," *ACM Trans. Comput. Syst.*, 23(3), 2005.

[195] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proc. ISCA-27*, 2000.

[196] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proc. HPCA-4*, 1998.

[197] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the Oklahoma update," *IEEE Parallel Distributed Technology*, 1993.

[198] H. Sundell and P. Tsigas, "Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems," in *Proc. IPDPS*, 2003.

[199] G. J. Sussman and G. L. Steele Jr, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, 11(4), 1998.

[200] A. S. Tanenbaum and D. J. Wetherall, *Computer networks*, 5th ed., P. Hall, Ed., 2010.

[201] J. E. Thornton, *Design of a Computer—The Control Data 6600*. Scott Foresman & Co, 1970.

[202] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[203] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, Jan. 1967.

[204] M. Tremblay and S. Chaudhry, "A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor," in *Intl. Solid-State Circuits Conf.*, 2008.

[205] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. SOSP-24*, 2013.

[206] N. Vachharajani, "Intelligent speculation for pipelined multithreading," Ph.D. dissertation, Princeton University, 2008.

[207] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proc. PACT-16*, 2007.

[208] A. Varga and A. Şekercioğlu, "Parallel simulation made easy with OMNeT++," in *Proc. ESS*, 2003.

[209] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy, "NePalTM: Design and implementation of nested parallelism for transactional memory systems," in *ECOOP*, 2009.

[210] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. ISCA-19*, 1992.

[211] D. Wall, "Limits of instruction-level parallelism," in *Proc. ASPLOS-IV*, 1991.

[212] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, 10(1):85–86, 1967.

[213] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the Tile Processor," *IEEE Micro*, 27(5), 2007.

[214] M. Wimmer, F. Versaci, J. Träff, D. Cederman, and P. Tsigas, "Data structures for task-based priority scheduling," in *Proc. PPoPP*, 2014.

[215] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, 31(2), 2011.

[216] M. Xu, R. Bodik, and M. D. Hill, "A flight data recorder for enabling full-system multiprocessor deterministic replay," in *Proc. ISCA-30*, 2003.

[217] C. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *ICDCS*, 1988.

[218] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proc. PASCO*, 2007.

[219] 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 *Proc. HPCA-13*, 2007.

[220] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *Proc. SPAA*, 2013.

[221] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proc. SPAA*, 2008.

[222] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," in *Proc. HPCA-5*, 1999.

[223] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, "GraphIt - A High-Performance DSL for Graph Analytics," *CoRR*, abs/1805.00923, 2018.