A Hardware and Software Architecture for Pervasive Parallelism

by

Mark Christopher Jeffrey

Bachelor of Applied Science, University of Toronto (2009)
Master of Applied Science, University of Toronto (2011)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author ....
Department of Electrical Engineering and Computer Science
October 25, 2019

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

A Hardware and Software Architecture for Pervasive Parallelism

by

Mark Christopher Jeffrey

Submitted to the Department of Electrical Engineering and Computer Science
on October 25, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Parallelism is critical to achieve high performance in modern computer systems. Unfortunately, most programs scale poorly beyond a few cores, and those that scale well often require heroic implementation efforts. This is because current parallel architectures squander most of the parallelism available in applications and are too hard to program.

This thesis presents Swarm, a new execution model, architecture, and system software that exploits far more parallelism than conventional multicores, yet is almost as easy to program as a sequential machine. Programmer-ordered tasks sit at the software-hardware interface. Swarm programs consist of tiny tasks, as small as tens of instructions each. Parallelism is dynamic: tasks can create new tasks at run time. Synchronization is implicit: the programmer specifies a total or partial order on tasks. This eliminates the correctness pitfalls of explicit synchronization (e.g., deadlock and data races). Swarm hardware uncovers parallelism by speculatively running tasks out of order, even thousands of tasks ahead of the earliest active task. Its speculation mechanisms build on decades of prior work, but Swarm is the first parallel architecture to scale to hundreds of cores due to its new programming model, distributed structures, and distributed protocols. Leaning on its support for task order, Swarm incorporates new techniques to reduce data movement, to speculate selectively for improved efficiency, and to compose parallelism across abstraction layers.

Swarm achieves efficient near-linear scaling to hundreds of cores on otherwise hard-to-scale irregular applications. These span a broad set of domains, including graph analytics, discrete-event simulation, databases, machine learning, and genomics. Swarm even accelerates applications that are conventionally deemed sequential. It outperforms recent software-only parallel algorithms by one to two orders of magnitude, and sequential implementations by up to 600× at 256 cores.

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

Acknowledgments

This PhD journey at MIT has been extremely rewarding, challenging, and stimulating. I sincerely thank the many individuals who supported me along the way.

First and foremost, I am grateful to my advisor, Professor Daniel Sanchez. His breadth of knowledge and wide range of skills astounded me when I arrived at MIT and continue to impress as I depart. My background was in parallel and distributed systems, and Daniel quickly brought me up to speed on computer architecture research. Early on, Daniel opined that a lot of fun happens at the interface, where you can change both hardware and software to create radical new designs. Now, I could not agree more. Daniel taught me how to think about the big picture challenges in computer science, then frame tractable research problems, and use limit study prototypes to quickly filter ideas as promising or not. He helped me improve my communication of insights and results to the broader community. He also taught me new tricks to debug low-level protocol deadlocks or system errors. Daniel exemplified strong leadership: he would recognize and nurture particular strengths in his students, and rally strong teams to tackle big important problems, while encouraging us to work on our weaknesses. Daniel both provided so much freedom throughout my time at MIT, yet was always available when I needed him.

I would like to thank my thesis committee members, Professor Joel Emer and Professor Arvind. Over the years, Joel taught me the importance of carefully distilling the contributions of a research project down to its core insights; implementation details are important, but secondary to understanding the insight. Throughout my career, I will strive to continue his principled approach to computer architecture research, and his welcoming, inclusive, and empathetic approach to mentorship. Arvind, who solved fundamental problems in parallel dataflow architectures and languages, provided a valuable perspective and important feedback on this work.

This thesis is the result of collaboration with an outstanding team of students: Suvinay Subramanian, Cong Yan, Maleen Abeydeera, Victor Ying, and Hyun Ryong (Ryan) Lee. Suvinay was a crucial partner in all of the projects of this thesis. I appreciate the hours we spent brainstorming, designing, and debugging. I learned a lot from his persistence, deep focus, and unbounded optimism. Cong invested a few months to wrangle a database application for our benchmark suite. We learned a lot from such a large workload. Maleen improved the modeling of our simulations, implemented several applications, and, bringing his expertise in hardware design, provided valuable feedback and fast prototypes to simplify our designs. Victor added important capabilities to our simulation in the Espresso and Capsules project, identified several opportunities for system optimization, and has been a crucial sounding board for the last three years. I am excited to see where his audacious work will lead next. Ryan implemented new applications that taught us important lessons on how to improve performance.

I am thankful to the members of the Sanchez group: Maleen Abeydeera, Nathan Beckmann, Nosayba El-Sayed, Yee Ling Gan, Harshad Kasture, Ryan Lee, Anurag Mukkara, Quan Nguyen, Suvinay Subramanian, Po-An Tsai, Cong Yan, Victor Ying, and Guowei Zhang. In addition to providing excellent feedback on papers and presentations, they brought fun and insight to my time at MIT, through technical discussions, teaching me history, eating, traveling to conferences, and hiking the trails of New York and New Hampshire. Thanks to my officemate, Po-An, who shared much wisdom about mentorship, research trajectories, and career decisions. I thoroughly enjoyed our one official collaboration on the MICRO-50 submissions server, and our travels around the globe.

I appreciate my dear friends across Canada and the United States, who made every moment outside the lab extraordinary.

Last but not least, an enormous amount of thanks goes to my family. My siblings encouraged me to start this PhD journey and provided valuable advice along the way. My parents boosted morale with frequent calls full of love and encouragement and relaxing visits back home. My nieces and nephews brought a world of fun and adventure every time we got together. I got to watch you grow up over the course of this degree. Finally, a very special thanks goes to my wife, Ellen Chan, who brings joy, fun, and order to my life. I could not have finished this thesis without her love and support.

I am grateful for financial support from C-FAR, one of six SRC STAR-net centers by MARCO and DARPA; NSF grants CAREER-1452994, CCF-1318384, SHF-1814969, and NSF/SRC grant E2CDA-1640012; a grant from Sony; an MIT EECS Jacobs Presidential Fellowship; an NSERC Postgraduate Scholarship; and a Facebook Fellowship.

Contents

CHAPTER 1

Introduction

This thesis focuses on hardware-software co-design to exploit challenging types of parallelism. Through emerging workloads and an explosion of data production, users demand ever higher compute performance from datacenters in the cloud [36] to devices at the edge [334]. Historically, the microprocessor industry met performance demand, by leveraging Moore's Law [261] and Dennard scaling [103] to transparently and exponentially improve sequential software performance. Circuit designers exploited the constant power density of transistor scaling to increase switching frequency. Computer architects used the growing hardware parallelism from increasing transistor counts to extract instruction-level parallelism (ILP) from sequential programs, by finding independent instructions to process simultaneously. Unfortunately, by the mid 2000s these approaches were yielding diminishing returns and computer systems became highly energy-constrained [58]. Transparent performance improvements would no longer come to sequential software [9, 153, 279, 357].

To continue performance scaling, we must reduce energy per operation, and parallelism paves all roads ahead. Modern technology yields chips with several billion transistors [122, 159, 249, 310, 319], which can be assembled into thousands of functional units, like ALUs and FPUs. Ideally, an application would use these units simultaneously, scaling its performance by exploiting the abundance of hardware parallelism, while reducing energy consumption with lower transistor switching frequencies [74, 118, 189, 331]. The challenge then is how to effectively utilize these units: how should software *express* the parallelism in an application, and what support should hardware provide to efficiently *extract* the parallelism? Put another way, *what execution models and architectures do applications require to execute thousands of operations per cycle?* Although the number of transistors on a chip is now plateauing [387], pursuing this question remains critical as modern computer systems fail to bring the parallelism benefits of modern technology to the vast majority of application domains.

At one extreme, imperative sequential execution is so intuitive and general that it has remained the prevalent form of computing for over 70 years [385]. Every program can be implemented as an ordered sequence of instructions. However, no practical superscalar out-of-order uniprocessor is likely to ever extract sufficient ILP to keep tens of functional units well utilized [179], let alone thousands.

At another extreme, GPUs [2, 148] and other domain-specific accelerators [78, 206, 373] successfully execute hundreds to thousands of operations in parallel. GPUs specialize the microarchitecture to exploit data-level parallelism through vector processing, and accelerators specialize the data path and optimize data movement for particular application domains. However, their execution models are specialized and restricted, supporting only those few domains with abundant easy-to-express parallelism. Moreover, accelerators are only viable for the few high-value applications that can recoup their considerable engineering costs; the economics of many applications require a versatile and general-purpose approach to parallelism.

Between these two extremes are multicores, which simultaneously execute multiple instruction sequences, called *threads*, across tens [232, 249, 319, 361] to hundreds [101, 335, 343] of cores or hardware contexts. Permitting arbitrary code and possibly disjoint threads, the multithreaded execution model is more flexible than that of GPUs and accelerators. Multicores strive to keep functional units busy by extracting parallelism both across threads (thread-level parallelism) and within each thread (instruction-level parallelism). However, even after decades of research in parallel computing, multithreaded programs that scale beyond even a few cores remain exceptional [127, 185]. Most applications are left sequential and on a path to stagnant performance. Even this general-purpose execution model can only express a small fraction of the parallelism available in applications, and is too hard to program [226]; multicores do not provide sufficient architectural support for easy-to-use, large-scale parallelization. To make abundant parallelism pervasive, *this thesis enhances the execution model and microarchitecture of shared-memory multicores to express and extract challenging types of parallelism*.

1.1 Challenges

Fundamentally, expressing the parallelism in a program consists of two steps: *(i)* dividing the work into *tasks* that may run concurrently, and *(ii)* specifying some order of execution among tasks with potential data dependences to ensure correct behavior. The system extracts the parallelism by running tasks simultaneously, while ensuring that data flow in the specified order. Unfortunately, modern multicores provide limited architectural support in both dimensions, and therefore fail to efficiently extract parallelism for applications that are challenging in either. To exploit abundant parallelism in a broad range of applications, two fundamental challenges remain to be solved.

Multicores lack general and efficient support to enforce order: When executed sequentially, an application's tasks run in a specific order—its sequential program order. However, the application semantics may permit one, few, or many correct orderings of its tasks. In fact, it is the order of data-dependent tasks that affects program output. Therefore, when executed concurrently, the tasks' dependent data accesses must be carefully ordered, or *synchronized*, so that the result of their execution matches one of those correct task orders. For example, the signal toggling events of a digital circuit simulation must execute in order of simulated time [144]. In contrast, the *serializability* [285] of database transactions is more flexible, permitting any interleaving of data accesses that is equivalent to any serial order of transactions. In some applications, all data dependences are known *statically* at compile time (e.g., dense linear algebra and multimedia streaming). The ordering of such *regular data dependences* can be enforced efficiently by devising a parallel dataflow task schedule [43, 104, 227] offline. However, many other applications are plagued by dependences that manifest *dynamically* (e.g., pointer chasing and graph analytics). Only an oracle can devise a task schedule in advance that respects the programmer's desired order of these *irregular data dependences*, while running independent tasks in parallel. Absent an oracle, real systems rely on either pessimistic or optimistic run-time synchronization tactics.

Pessimistic synchronization mechanisms include locks, barriers, condition variables, etc. The programmer uses these tools to constrain the ordering of instructions among concurrently executing threads, to maintain the integrity of program state by ensuring that a valid task ordering is retained. Unfortunately, even if irregular dependences among tasks rarely manifest, they are unknown in advance, so this pessimistic approach requires conservative and frequent synchronization that scales poorly beyond a few cores [403]. Moreover, synchronization using these tools has been the bane of parallel programming, with pitfalls such as deadlock, data races, and other concurrency bugs [226, 240]. Non-blocking synchronization [180] addresses the scalability challenges of locking [195, 254], but is arguably even more difficult to use correctly [358].

To address these limitations, optimistic mechanisms *speculatively* execute tasks in parallel to find a valid task schedule. This approach detects unsafe instruction interleavings and undoes the effects of offending tasks, to retrospectively guarantee that the execution result matches some programmer-approved task order. As software-only techniques for speculation incur non-negligible overheads [67, 176], hardware support for transactional memory (TM) [182] and thread-level speculation (TLS) [344] has received considerable attention in research. Several commercial multicores now implement hardware TM [20, 64, 198, 389, 401]. Unfortunately, while TM and TLS are simpler to use than locking, their execution models still restrict the types of parallelism that the programmer can express. TM only guarantees task atomicity and isolation, so the programmer has no ability to express parallelism subject to a particular task order. TLS parallelizes sequentially ordered code, but this unnecessarily constrains the parallelism expressed in many applications. Additionally, previously proposed TLS implementations suffer from mechanisms that scale poorly beyond a few cores (Section 2.4.2). Ideally, speculation should extract the task-level parallelism that only an oracle can see in advance. In practice, it remains unclear what execution model will grant programmers flexibility in expressing their desired task order(s), and what supporting architecture can extract enough parallelism to effectively utilize modern chips.

Multicores lack support for scheduling short tasks: Multicores are amenable only to algorithms that consist of *(i)* tasks known *statically* at compile time or *(ii)* coarse-grain tasks of thousands to millions of instructions that are created *dynamically* at run time. However, many algorithms are more naturally expressed using dynamic fine-grain tasks of a few tens to hundreds of instructions, and exploiting this fine-grain parallelism is often the only way to make these algorithms scale. Unfortunately, because the multithreading interface is task-agnostic, dynamic tasks are managed in software data structures, so short tasks cause large overheads that overwhelm the benefits of parallelism [90,176,177,215,312,409].

To amortize these overheads, techniques like bulk-synchronous parallelism [377] coalesce short tasks into batches of many thousands of instructions. Unfortunately, this coarsening thwarts the programmer's expression of parallelism, reverting thousands of short potentially-parallel tasks into a few long sequential tasks. Furthermore, it is not safe to arbitrarily coalesce tasks in applications with complex ordering constraints that schedule tasks to run far in the future (Chapter 3).

Although some architectures [161,221,326] provide hardware schedulers for short *non-speculative* tasks, their execution models remain hard to use for many classes of applications, due to the limited support for synchronization, discussed previously. Hardware management of speculative tasks requires new architectural techniques. What should be the design of hardware task queues when resources are held longer to detect invalid instruction interleavings? How should hardware task resources be allocated when a task's very existence is speculative, i.e., when it can be created speculatively?

Summary: For the small minority of applications with the right task and dependence structures, multicores and other parallel architectures enable programmers to improve performance by exploiting "the easiest form of parallelism that matches the domain" [179]. In contrast, this thesis focuses on general-purpose support for the heavy tail of applications for which current systems force programmers to leave parallelism on the table. Amdahl's Law makes it clear that targeting only the easy parallelism yields diminishing returns [15,185]. Current execution models and architectures choke performance with sequential bottlenecks that leave modern chips grossly underutilized.

1.2 Contributions

This thesis presents[1] a new execution model, multicore microarchitecture, and cross-layer techniques that combat sequential bottlenecks, by easily expressing and efficiently extracting types of parallelism that were previously inaccessible. The resulting system, called Swarm, supports fine-grain tasks with low overheads and provides efficient, easy-to-use synchronization. Swarm achieves this by exposing programmer-defined *task order* as a simple, implicit, and general form of synchronization. Partially ordered tasks capture the ultimate goal of complex synchronization patterns—including mutual exclusion, signaling, and barriers—yet are arguably almost as easy to use as sequential programming. With this new software-hardware interface, Swarm programs can convey not only a sequential task order, but also partial orders, and even more nuanced order constraints, such as creating tasks in a different order than they must run. Applications with the latter constraints, so-called *ordered irregular algorithms* [289], have been extremely challenging to parallelize on current systems.

Swarm programs consist of short, dynamically created tasks that are ordered by programmer-defined timestamps. Swarm hardware extracts parallelism by speculatively running available tasks out of order, even thousands of tasks ahead of the earliest speculative task. The microarchitecture scales to large core counts due to distributed structures and speculation mechanisms.

Perhaps surprisingly, Swarm's efficient support for an ordered-task interface subsequently enables simple solutions to further push the scalability, efficiency, and generality of the system. First, task order enables us to increase scalability by breaking tasks more finely than ever before and *sending compute close to the data that it accesses* to improve locality. Second, we enhance the system to *combine speculative and non-speculative parallelism*, executing speculatively only when required to scale, and otherwise improving efficiency with non-speculative execution. Finally, task order enables *seamless composition of parallel algorithms*, improving the generality of the system and opening the door to parallelizing large complex programs.

Combining these techniques, Swarm achieves near-linear scaling to 256 cores on a diverse set of 20 applications that span domains such as graph analytics, machine learning, databases, simulation, and genomics. The Swarm architecture is evidently on a promising path toward enabling a broad range of applications to harness the massive parallelism of modern technology.

Mining parallelism from ordered tasks: *Swarm* [202, 203] is a new execution model and microarchitecture with ordered tasks at the software-hardware interface. A Swarm program is expressed as a set of short, timestamped tasks. Each task may create new children tasks for later execution, usually deriving their timestamps from local state. Tasks appear to execute in timestamp order; those with equal timestamp appear

[1] The ideas, designs, and experiments in this work were developed collaboratively over many years with six graduate students. Each chapter details the specific contributions of this thesis in detail.

atomic.

Swarm hardware enhances a tiled multicore. It is optimized to support tasks as small as tens of instructions. It extracts parallelism by speculatively executing available tasks out of order, while committing them in order. Doing this efficiently requires overcoming several hurdles. How can we manage tiny tasks efficiently, keep cores busy with useful work, perform scalable order-aware conflict detection, and commit many tasks in order in a scalable fashion? Prior work solved some of these problems at small scale with centralized tracking structures. Swarm is the first system to solve these problems using distributed techniques that are amenable to multicores with hundreds of cores. For example, whereas prior work would abort all future tasks upon a misspeculation, Swarm *selectively aborts* descendant and data-dependent tasks with scalable tracking structures. As a result of our novel contributions, Swarm *efficiently speculates thousands of tasks ahead of the earliest speculative task*.

Exploiting locality: Parallelism must not come at the expense of locality. Systems that support speculative parallelization—like Swarm, TM, or TLS—can uncover abundant parallelism in challenging applications. However, they scale poorly beyond tens of cores because, unlike their non-speculative counterparts, they *do not exploit the locality available in speculative programs*. At first glance, locality and speculation seem to be at odds: exploiting locality requires knowing the data accessed by each task, but a key advantage of speculation is precisely that one need not know the data accessed by each task. However, with tiny ordered tasks at the interface, we find that in fact most of the data accessed is known at run time, just before a task is created.

Building on this insight, we extend Swarm with *spatial hints* [201], a technique to enable software to convey locality to hardware, to run tasks likely to access the same data in the same place. This exploits cache locality and avoids global communication. When a new task is created, the programmer provides a *spatial hint*, an integer that abstractly denotes the data that the task is likely to access. Hardware maps tasks with the same hint to the same tile to localize data accesses, and balances load by moving hints, and their tasks, around the system. For example, by leveraging task order, many graph algorithms can be restructured so that each task operates on a single vertex; using the vertex ID as a hint sends tasks for the same vertex to the same place, reducing data movement. Swarm with spatial hints empowers programmers with a semantically rich execution model to scale programs to hundreds of cores by exposing even *more opportunities to exploit locality than in non-speculative architectures*. At 256 cores, hints achieve near-linear scalability and outperform Swarm's baseline random task mapping by up to 16×. Hints also make speculation more efficient, reducing wasted work by 6.4× and network traffic by 3.5× on average.

Mixing speculative and non-speculative parallelism: Parallel systems should support both speculative and non-speculative parallelism. Even applications that need speculation to scale have some work that is best executed non-speculatively. Consider three examples. First, some tasks are well synchronized and running them speculatively adds overhead and needless aborts. Second, other tasks need speculation to scale but perform short actions, such as memory allocation, whose dependences are best hidden from hardware because they commute. Finally, non-speculative parallelism is required to perform irrevocable actions, such as file or network I/O, in parallel. Work on hardware transactional memory has considered these issues, but does not support programmer-controlled ordered tasks. All systems for ordered parallelism, including Swarm, disallow non-speculative parallelism.

Espresso and *Capsules* [204] bring the benefits of non-speculative execution to the Swarm architecture. Espresso is an expressive execution model that generalizes Swarm timestamps and spatial hints; it efficiently coordinates concurrent speculative and non-speculative ordered tasks. Espresso lets the system decide whether to run certain tasks speculatively or non-speculatively, *reaping the efficiency of non-speculative parallelism when it is plentiful, while exploiting speculative parallelism when needed to scale*. Capsules bring ideas from OS system calls and virtual memory to speculative architectures. They enable speculative tasks to safely transition out of hardware-managed speculation and protect certain memory regions from speculative accesses. Espresso outperforms Swarm's speculative-only execution by up to 2.5×. Capsules enable important system services, like a speculation-friendly memory allocator that improves performance by up to 69×. Espresso and Capsules give programmers the option to improve the efficiency of the ordered-tasks interface.

Composing parallelism: Large complex programs can have large blocks that must appear atomic. Left sequential, these blocks become performance bottlenecks. For example, a database transaction could be millions of cycles long. Database designers target parallelism among sequential transactions, but *there is plentiful parallelism nested within each transaction* if we break up each query and update into its own ordered task. But how do we ensure that all of the tasks representing one transaction appear atomic with respect to those of other transactions?

Fractal [352] is an execution model that pushes the generality and expressivity of the Swarm interface by capturing more complex synchronization and ordering constraints with nested parallelism. Examples include composition (i.e., calling an ordered algorithm from within an ordered algorithm), or running independent ordered algorithms concurrently. Fractal groups tasks into *domains*, with tasks in each domain executing in the domain's timestamp order. Tasks can create domains and set optional order constraints among domains. This carefully designed interface virtualizes domains cheaply, allowing an unbounded number of domains in software with simple hardware support. Fractal accelerates large complex workloads by up to 88×.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides relevant background and summarizes related work. Chapter 3 presents the design and evaluation of the

Swarm architecture. Building on Swarm's support for task order, Chapter 4, Chapter 5, and Chapter 6 show how to generalize the execution model and microarchitecture to exploit locality with spatial hints, speculate selectively with Espresso and Capsules, and compose speculative parallelism with Fractal. Finally, Chapter 7 concludes the thesis and proposes future work.

CHAPTER 2

Background

Expressing the parallelism in a program, or *parallelizing* it, consists of two fundamental steps: dividing the work into tasks that will run concurrently, and enforcing some order of execution among those tasks that may be data dependent. Any given execution model and its supporting architecture will enable or constrain these steps, affecting the scalability and performance of applications to varying extents. Currently, only a minority of applications have a path to high performance, as modern computer systems constrain the types of parallelism that are efficiently expressed. To unlock abundant parallelism from the majority of applications, computer systems must support even the most challenging types of parallelism with scalable hardware implementations.

The shared-memory multicore architecture, the baseline for this thesis, has been commercially lucrative in part because its microarchitecture scales well: replicating a core design across the chip reduces engineering costs [37, 279]. Figure 2-1 illustrates a tiled homogeneous design. Every tile has several cores, each core runs a thread of

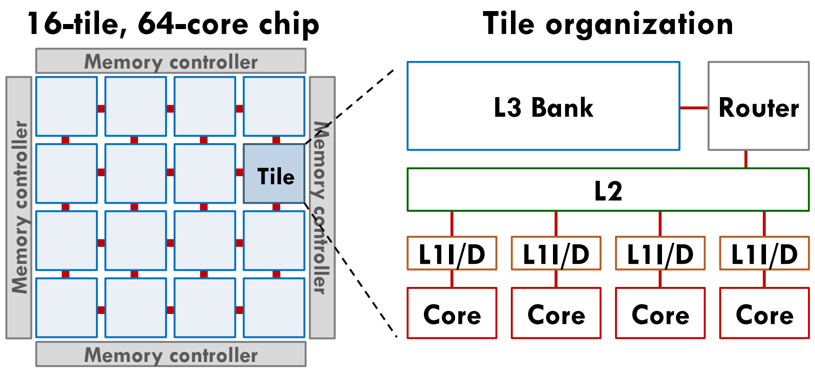


Figure 2-1: An example tiled homogeneous 64-core multicore.

instructions, and tiles are distributed across the chip, communicating through an on-chip network [126]. To reduce average memory access latency, this system uses a three-level coherent [345] cache hierarchy [32]. Unfortunately, while multicore hardware scales well, its multithreaded execution model constrains the types of parallelism that can be efficiently expressed, stymieing the efforts of programmers.

The rest of this chapter discusses the opportunities, techniques, and challenges in exploiting different types of parallelism. Section 2.1 describes our view of the types of parallelism in applications. In particular, it identifies how three properties of an application's tasks affect the degree of run-time overhead required to safely express and extract parallelism. The next sections provide an overview of existing software and hardware techniques to exploit these types of parallelism, beginning with the simpler types in Section 2.2. Section 2.3 and Section 2.4 then summarize two broad approaches to exploiting the more challenging types of parallelism, highlighting where they fall short.

2.1 Important Properties of Task-Level Parallelism

An application may run through different phases, algorithms, or components. We distinguish three properties of an algorithm's tasks that influence how easily parallelism is expressed and extracted during that phase: *regularity*, *ordering*, and *granularity*. The first property describes how much information is known at compile time, which affects the resources required to orchestrate tasks at run time. The second describes how much flexibility the application permits in reordering tasks, in some cases reducing run-time overheads. The third describes the typical size of the tasks, impacting the relative cost of run-time overheads, and therefore the need for architectural support.

A *task* is an instance of some sequential block of code with input *arguments* that may read and write shared memory, and possibly perform externally visible actions like I/O and system calls. All instances of tasks might be known *statically* at compile time, or tasks might discover new work and create new tasks *dynamically* at run time. A *data dependence* exists between two tasks if they both access the same datum and at least one of them writes to the datum. The two tasks are *data-dependent* if they share at least one data dependence.

2.1.1 Task Regularity

An algorithm is *regular* when all of its tasks and their data dependences are known statically. Example applications include dense linear algebra, data streaming in image and media processing pipelines, and stencil computations. The tasks often operate on data structures like dense arrays and tensors using regular strides, and have predictable control flow. In the special case that all tasks are identical and operate on disjoint data, they can safely run in parallel during this *data parallel* phase [186]. In the general case, a compiler, runtime system, or programmer can express the *regular parallelism* by devising a schedule where independent tasks run in parallel, or by efficiently synchronizing the *regular data dependences* with techniques like static dataflow scheduling [43, 104, 227].

If any tasks or their data dependences are dynamic and therefore unknown a priori, the algorithm is *irregular*. Newly created tasks are scheduled for later execution, requiring run-time-managed queues, perhaps as part of an unbounded loop or a divide-and-conquer algorithm. *Irregular data dependences* arise among memory accesses through statically unknown pointers, for instance to data structures like lists, trees, and graphs. Given two such accesses, a dependence may or may not manifest, because the addresses depend on run-time values. Irregular algorithms are increasingly common in domains such as graph and network analytics [123], machine learning and recommendation systems [184], in-memory OLTP databases, and discrete-event simulation [144]. Because the dependences are unpredictable and new work may be discovered on the fly, the *irregular parallelism* of such an algorithm cannot be scheduled in advance, but must be safely found at run time.

2.1.2 Task Ordering

Although the sequential execution of an algorithm runs its tasks in one particular order, application-specific semantics may permit one, a few, or many correct orderings of the tasks. An *unordered* algorithm [289] is the most flexible, permitting tasks to be executed in any serial order. For example, OLTP databases have this property [285], along with the tasks within each superstep of an iterative graph algorithm such as connected components or PageRank [250, 338]. In contrast, an *ordered* algorithm [289] requires that tasks execute in some total or partial priority order, such as Kruskal's algorithm [216] to compute a minimum spanning tree on a weighted graph, which processes edges in increasing order of weight.

In particular, it is differing orders of data-dependent tasks that may change the result of a program. Two data-dependent tasks are *ordered* if the semantics require that all of their shared data flow from the writes of the earlier-ordered *predecessor* task to the reads of the later-ordered *successor* task, and not the other direction. The two tasks are otherwise *unordered* if either task can be precede the other; we assume that the tasks should be *atomic*, i.e. the data flow between the tasks must have no cycles. Possible reasons the programmer would permit unordered tasks include commutative operations, e.g., two tasks increment a shared counter but do not read it, or *don't-care non-determinism* [112, 289], i.e., the application permits more than one correct output.

Whether expressing the parallelism among ordered or unordered tasks, the interleavings of data-dependent accesses must be carefully *synchronized* to correspond to some permitted task order. However, the ability to reorder unordered tasks grants flexibility to compile-time or run-time systems when arbitrating access to shared data: the larger set of correct task orders permits a larger set of legal data-access interleavings.

Exploiting *unordered parallelism* may allow shorter run-time stalls, releasing run-time resources more quickly, or coarsening several tasks together [176], making *ordered parallelism* more challenging to exploit efficiently.

2.1.3 Task Granularity

Compilers, runtimes, programmers, and even hardware delineate task boundaries. Common candidates include loop iterations or function calls, which could appear at shallow or deep levels of control flow nests. One might select even coarser or finer *granularities*, or sizes, of code. Some tasks can be as short as a single instruction (such as in instruction-level parallelism), but we regard most *fine-grain* tasks as tens to thousands of instructions, and *coarse-grain* tasks as thousands to millions of instructions. Since large programs consist of layers of interacting components (e.g., a function that contains nested loops or calls into other libraries), it can be desirable to *compose nested parallelism*: invoking a parallelized algorithm from within another parallelized algorithm. We consider nested parallelism in Chapter 6.

Task granularity presents a performance trade-off. On one hand, expressing many fine-grain tasks can yield far more parallelism than few coarse-grain tasks, particularly when compounding the nested parallelism across layers of the program. On the other hand, the costs of fine-grain tasks are more pronounced, including task creation, queuing, synchronization, and other communication. So the selection of task granularity influences the type of architectural support required.

2.1.4 Open Opportunity: Fine-Grain Ordered Irregular Parallelism

Combined together, the regularity, order constraints, and granularity of an application's tasks impact the run-time costs required to safely extract its parallelism. At the lighter end of these dimensions, the schedule of regular tasks is optimized offline to efficiently run them in parallel, unordered tasks can be reordered to enable more compile- or run-time optimizations, and coarse-grain tasks often amortize any remaining run-time costs. At the heavy end, irregular tasks may be dynamically synchronized and queued in run-time structures, ordered tasks restrict the number of acceptable parallel schedules, and fine-grain tasks may do little useful work relative to their run-time costs. Current architectures have left a major opportunity for parallelism unexploited, and therefore have left many applications behind.

On one hand, the multithreaded execution model—long instructions sequences that synchronize through atomic instructions—and its backing multicore architectures provide limited architectural support along each of these three dimensions. They ultimately target the easier types of parallelism. For example, while regular task schedules can be mapped to parallel threads, dynamically created irregular tasks require memory-backed, software-managed queues that incur high overheads relative to a short task's useful work. This has restricted the domain of multicores to exploiting regular parallelism and coarse-grain irregular parallelism with infrequent synchronization (The following sections summarize existing parallelization techniques).

On the other hand, superscalar out-of-order uniprocessors provide architectural support to extract instruction-level ordered irregular parallelism from sequential programs. Unfortunately the granularity of instruction-sized tasks has proven too small to achieve our goal of keeping thousands of functional units well utilized (Section 2.4.1).

In contrast, this thesis focuses on the unexploited taxing end of these three dimensions: providing architectural support to extract the parallelism among fine-grain ordered irregular tasks. It is important to note that we do not propose niche solutions: targeting the most challenging case results in a versatile and general-purpose architecture that inclusively supports *more* types of parallelism and *more* application domains. For instance, accelerating task management for fine-grain tasks need not preclude running coarse-grain tasks. Support for irregular data dependences can be bypassed for regular data dependences, to run more efficiently. Any support for totally ordered or partially ordered tasks can still execute unordered tasks correctly and efficiently. Over the course of this thesis, we build a general execution model and architecture that, in exchange for only modest area and communication costs over conventional multicores, yields substantial returns by extracting abundant fine-grain ordered irregular parallelism to exploits the performance potential of modern chips.

2.2 Exploiting Regular Parallelism

The task dependence graph is a helpful abstraction to reason about, and in some cases express, task-level parallelism. A vertex represents a task that must run atomically. A directed edge represents a dependence from a predecessor task to its successor task, due to either *(i)* flow of data or *(ii)* a control dependence, e.g., a *parent* task creates a *child* task or a control instruction jumps to its target. The graph is directed and acyclic, or else it would not correspond to some serial task order. One can construct the task graph following the execution of an algorithm for a posteriori analysis, but a key distinction between regular and irregular algorithms is whether the task graph can be determined at compile time. This knowledge informs what techniques are used to exploit regular parallelism (discussed in this section) and irregular parallelism (Section 2.3 and Section 2.4).

A task is safe to execute once all of its predecessors in the graph have completed, and a group of safe tasks is the source of parallelism [314]. The statically known tasks and dependences of regular algorithms can enable techniques to devise parallel task schedules, offline, that express regular parallelism. The a priori knowledge is exploited to reduce the run-time overheads of synchronization and task scheduling.

Several programming languages and models facilitate the expression of regular parallelism. Data parallelism is the target of NESL [48], ZPL [235], and OpenMP [95] constructs like `parallel-for`. Some languages and models exploit the straightforward dataflow in particular domains like streaming [364], image processing pipelines [298], machine learning [4, 71], and digital signal processing [43, 228]. Others constrain the expression of parallelism for particular systems like GPUs [246], reconfigurable architectures [213], and distributed systems [3, 102].

Compilers construct and analyze a program dependence graph [133], a task graph at the granularity of individual instructions, to find regular parallelism. Compilers have some success parallelizing sequential code that operates on regular data structures [34, 256], such as “vectorizing" data-parallel inner loops. The polyhedron model optimizes parallelism and data locality in regular loop nests [30, 59, 132, 160]. Points-to analysis [125, 187, 190] and shape analysis [155, 178] can identify independent tasks that access some irregular data structures like trees, and can support expression of pipeline parallelism [281].

Although the set of task instances is known in advance, some software runtimes queue more tasks than there are cores to dynamically balance load across the system [51, 290]. Exploiting static knowledge, runtimes may *coarsen* several tasks into a large chunk to reduce queuing and synchronization overheads [215, 290, 374, 409].

Architectural support further reduces overheads for different patterns of regular parallelism. The single-instruction multiple-thread (SIMT) execution model is exploited by GPUs [2, 148] to amortize instruction-processing overheads over a batch of identical tasks. Single-instruction multiple-data (SIMD) instructions (e.g., AVX [135]) or vector processors [317] enable software to express data parallelism at the instruction granularity. Very long instruction word (VLIW) architectures enable compilers to express parallelism among a small number of independent, static-latency instructions [89, 136, 332], when branches are highly predictable [287]. Spatial architectures with tiles of processing elements (PEs) enable the compiler to configure direct communication of values among the PEs [386], while systolic architectures limit configurability by further specializing data paths to move data among the PEs in a pipeline before returning a result to memory [206, 222].

2.3 Exploiting Non-Speculative Irregular Parallelism

Conceptually, the challenge in parallelizing irregular algorithms is that the structure of the task graph is revealed over the course of execution. For example, when a task finds new work and creates a child task, we have learned that a vertex was added to the graph; when a pointer’s address is finally resolved, we may learn of a new data dependence edge. With this evolving task graph, run-time techniques must safely find the irregular parallelism which imposes more overheads. Broadly, there are two approaches to control the ordering of dependent data accesses among concurrent tasks: speculative (Section 2.4) and non-speculative synchronization. Programmers use non-speculative techniques to preclude data-access interleavings that would not result in any correct task ordering [329].

Explicit primitives like locks and barriers synchronize unordered tasks (mutually exclusive critical sections) and ordered tasks (program phases), respectively. Unfortunately, since data dependences are unknown, these mechanisms are often specified conservatively which impairs scalability. For example, coarse-grain locking conceptually adds unnecessary dependence edges to the task graph, to synchronize the few true data dependences that will manifest at run time. A barrier is appropriate to order large groups of tasks before one another, but is too blunt for applications that require ordering a smaller number of tasks, as we discuss further in Chapter 3. Non-blocking synchronization [180] using atomic instructions (e.g., compare-and-swap) can lead to scalable parallel implementations [195, 337]. Unfortunately, correct, efficient, and scalable synchronization with blocking or non-blocking mechanisms is widely regarded as the purview of experts, due to their challenges and pitfalls such as deadlock, priority inversion, data races, order violations [240], non-determinism [226], and performance under contention [99].

Software libraries and domain-specific frameworks can raise the level of abstraction. Concurrent data structure libraries provide tuned concurrent implementations of unordered data structures (e.g., hash tables, sets, or bags) and ordered data structures (e.g., stacks, lists, FIFO queues, and priority queues) [254, 255, 260, 354, 355]. Unfortunately, even though the operations on a single object are atomic, tasks that update multiple objects must resort to conventional synchronization and all of its pitfalls. Among other uses, these concurrent structures may queue dynamically created tasks, scheduled for later execution. While unordered algorithms can tolerate reordering of tasks, in some cases their performance is improved with priority task ordering [271]. For such cases, relaxed concurrent priority queues can scale to high core counts by relaxing dequeue order [14, 231, 311, 393], but again, the programmer must synchronize other data structures by conventional means. Some frameworks exploit particular application structure to abstract away the multithreaded interface, handling sychronization and scheduling under the hood. A general-purpose example is Galois [289], while graph frameworks are programmed around vertex- and edge-updating tasks, and exploit the unordered parallelism within *(i)* each frontier-based round [250, 270, 338, 408] or *(ii)* each level of a wide partial order [107].

Dataflow models: In *dataflow* execution the availability of data drives execution [104, 105]. Whereas the prior discussion considered the task graph conceptually, several dataflow-based software systems construct or emulate a task dependence graph or dataflow graph at run time. They use dataflow execution to synchronize dependent tasks. Any vertex with no inbound edges (predecessors) represents a task that is safe to dispatch to a core. Typically, the system's underlying scheduler distributes such ready tasks to per-thread software-managed worklists to be run in parallel. When a task finishes, its vertices and outbound edges are removed from the graph, potentially adding other tasks to the ready worklists. The onus falls on the programmer to convey or constrain dependences among tasks, for instance, by declaring read and write sets (e.g., OpenMP 4.0 [54], Jade [312], SMPSs [288], OmpSs [120], XKaapi [152], and Gupta and Sohi [165]), by declaring explicit dependence edges (e.g., TBB [196] and Nabbit [11]) or by restricting a task's accesses to a subset of data (e.g., Cilk [142], Chapel [73], Legion [38], and X10 [76]). This just-in-time coordination [289] of tasks graphs can be viewed as a generalization of the inspector-executor model [321], which parallelizes irregular loops through two phases: the inspector phase identifies the dependences among iterations, and the executor phase uses this information to create a parallel execution schedule. Deterministic reservation [46] is a related strategy wherein *reserve* phases enable tasks to reserve priority-ordered access to data, and *commit* phases determine which tasks succeed. While some of the previous systems permit dynamic creation of tasks, they do not support ordered irregular parallelism in general, as newly created priority-ordered tasks would change the dependence structure of the underlying task graph. Kinetic dependence graphs [177] address this gap by extending traditional task dependence graphs with programmer-defined methods to *(i)* identify safe tasks to dispatch and *(ii)* specify how to update the task graph.

Although some software systems operate on a task dependence graph, the conventional representation of a dataflow program is a *dataflow graph* [104, 131]. These graphs have subtle differences. In dataflow graphs, each vertex represents an *operator* or function to be applied to incoming data. Every directed edge represents a channel for a datum to move from its producing operator to a consuming operator (a data dependence). Whereas a vertex in the task graph is a dynamic instance of some code (a task), a dataflow vertex represents the static code of the operator or function. A dataflow operator should have no side effects, which is why task-dependence systems require that data dependences in memory are made explicit. In dataflow graphs, an operator's consumed data must come from its inputs. Whereas a task graph is acyclic, the dataflow graph may have cycles, for example to represent loops [22]. Control is represented as boolean data that steers the flow of other data via special operators, or that predicates operator actions. Execution of the dataflow graph is data-driven [105]: an operator is ready to execute when data arrives at all its input edges, meaning that several operators could execute in parallel. Dataflow languages have a long history of work, surveyed by Johnston et al. [205].

Architectural support: While the previous software systems suffice for large tasks, the non-negligible overheads of task scheduling and queuing overwhelm the parallelism benefit of fine-grain tasks. For example, every interaction with a task worklist requires accesses to memory. While dequeues from a local worklist may hit in a core's local cache, remote enqueues and task redistributions induce cross-chip network traffic that hurt performance and energy use. Contreras and Martonosi characterize the performance of TBB [90] and recommend tasks of at least 100,000 instructions to amortize scheduer overheads. Similarly, Morais et al. [263] find that the Nanos software task scheduler [363] requires 100,000-cycle tasks to saturate an 8-core machine.

Classic dataflow architectures [105,380] accelerate dataflow scheduling, operating at the finest granularity where each operator is a single instruction. The architectures consists of interconnected processing elements. Every instruction identifies its consumer instructions so that when it executes, *tokens* can (conceptually) carry the output data to their targets. In practice, a hardware structure at each processing element holds and matches tokens that have arrived, and once all tokens for an instruction are present, the instruction is issued to a functional unit. While static dataflow architectures [105] permit only one instance of an operator (instruction), dynamic dataflow architectures [23], uncover more parallelism by permitting several instances of each operator through a set of mechanisms that *tag* each generated token with the instance of its target operator.

The more recent work in dataflow architectures has proposed hybrid dataflow/von-Neumann systems to exploit the parallelism of dataflow and the programability of the sequential model. TRIPS [62] is a tiled architecture that exploits dataflow parallelism within each large VLIW-like block of 128 instructions that encode a dataflow graph, while using imperative control flow between blocks. WaveScalar [359, 360] is a dataflow architecture that supports memory ordering to run imperative sequential and multithreaded code. Whereas the previous systems work at the instruction granularity, Task Superscalar [129], TDM [68], and Phentos [263] operate at the task granularity, providing hardware support for task dataflow scheduling or dependence analysis atop baseline multicores. Yazdanpanah et al. [397] survey hybrid dataflow architectures in more detail.

Whereas dataflow architectures reduce synchronization overheads, other hardware proposals reduce the overheads of dynamic task queuing and load balance in multicores. Carbon [221] extends a multicore with hardware task queues and a work-stealing protocol to distribute load across cores. ADM [326] enables cores to communicate by efficiently sending and receiving asynchronous, short messages, bypassing the memory hierarchy. This enables software-configurable task schedulers with low-overhead load balancing.

Limitations: Non-speculative parallelization techniques are amenable to large tasks, but the overheads of software task queuing and synchronization overwhelm the benefit of parallelism for short tasks on modern multicores. Architectural support reduces the overheads for some applications, but the prior hardware techniques do not support ordered irregular parallelism in general. In particular, algorithms that dynamically create tasks scheduled to run far in the future, such as the discrete-event simulation of a circuit, cannot be effectively expressed in the dataflow model. Moreover, non-speculative techniques place some amount of burden on the programmer to restrict data access patterns, explicitly synchronize tasks, or explicitly declare task dependences. Speculative parallelization (Section 2.4) reduces some of the burden on the programmer and can extract more parallelism when dependences are unknown.

2.4 Exploiting Speculative Irregular Parallelism

While non-speculative techniques pessimistically constrain the concurrent execution of tasks, speculative techniques optimistically run available tasks in parallel, and recover if program correctness would be violated. *Data-dependence speculation* frees programmers from explicitly managing data dependences, such as tagging tasks, identifying the read- and write-set of each task, or specifying *how* to synchronize. Instead, programmers simply declare *which* task orders are correct or which code blocks require (unordered) atomicity. Given this correctness specification, the underlying system executes tasks in parallel and *speculates* either *(i)* that all tasks are independent, or *(ii)* that for every task, all its data-dependent predecessors have finished. To find a safe task schedule, the software or hardware system automatically detects when this guess is wrong and recovers to a consistent state upon *misspeculation*. A *conflict* or *dependence order violation* arises between two tasks when their data-dependent accesses did not flow in a correct order. If no conflicts are detected, then the speculation was correct, so the tasks *commit* by making their updates *non-speculative*. However, if any speculative tasks conflict the system may need to *abort* some of them to undo their effects, then restart their execution.

Systems that employ speculative parallelization implement three basic primitives. *Version management* enables recovery from mispeculation by distinguishing speculative and non-speculative versions of data. *Conflict detection* tracks the data read and written by each task (its *read* and *write sets*) to detect order violations. *Conflict resolution*, upon a conflict, determines which task(s) should stall or abort to preserve correctness.

At the finest granularity, high-end commercial processors employ speculation to extract ILP. The sequential execution model specifies a unique correct order of instructions. Targeting particular microarchitectures, compilers optimize the instruction stream to better express the parallelism [328]. Architectural support for ILP finds independent instructions to process out of order and in parallel, through non-speculative dynamic dataflow analysis, speculating on the target of branches, and speculating on unresolved memory addresses (Section 2.4.1).

At coarse granularities, software systems extract speculative parallelism among large tasks. In transactional databases, optimistic concurrency control ensures transactions appear to run in any serializable order [223, 371]. In discrete-event simulation, Time Warp [200] ensures that events appear to execute in order of simulated time. General-purpose frameworks employ speculative parallelization of irregular loop iterations, while reducing conflict rates by exploiting semantic commutativity [220], semantic independence [65], pipeline parallelism [192, 303, 376], value-based conflict detection [113], and other techniques. Applications amenable to large tasks of thousands to millions of instructions have sufficient useful work to amortize the software overheads of speculation, including validation and commit. However, there is often far more fine-grain parallelism available with short tasks of tens to hundreds of instructions, but at this granularity, the overheads overwhelm any performance benefit of parallelism. Moreover, short *ordered* tasks exacerbate the problem, particularly when serially committing tasks in order, as it becomes the bottleneck [176, 220, 303, 376].

Architectural support for task-level speculation has been proposed to reduce these overheads, through two dominant execution models. Thread-level speculation (TLS) automatically extracts task-level parallelism from sequential programs [128, 344] and transactional memory (TM) performs optimistic synchronization of unordered transactions in multithreaded programs [171, 182]. Unfortunately, as we discuss in Section 2.4.2 and Section 2.4.3, neither execution model is sufficiently general to capture the ordered irregular parallelism latent in many important algorithms. Moreover, the speculation mechanisms underlying TLS hardware scale poorly beyond the small-scale systems of that era.

2.4.1 Dynamic Instruction-Level Parallelism

The sequential execution model is the simplest and longest-standing software-hardware interface [385]. Software provides a totally ordered sequence of instructions, where register- or memory-based data dependences flow in sequence order. From the view of the Section 2.1 taxonomy, this execution model exposes a form of ordered irregular parallelism that is restricted to a chain of single-instruction tasks, illustrated in Figure 2-2. An instruction's data dependences are identified dynamically, and every instruction is control-dependent on its predecessor, since a branch or jump instruction can appear at any position in the sequence.

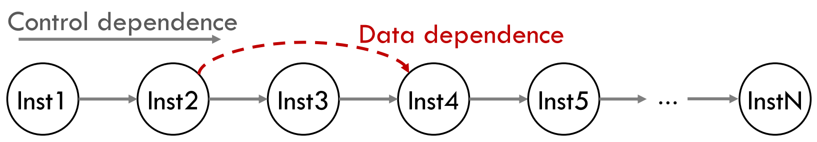


Figure 2-2: The sequential execution model consists of an ordered chain of control-dependent instructions, which limits expression of far-away parallelism. Instructions may also be data dependent, as `Inst4` is on `Inst2`.

Modern high-end processors extract dynamic instruction-level parallelism (ILP) from the ordered stream, by processing independent instructions simultaneously using pipelining, register renaming, out-of-order execution, and superscalar execution [305,342]. While data dependences on registers are synchronized non-speculatively using dataflow scheduling [286], these processors also employ two forms of speculation to find more parallelism. *Control-flow speculation* predicts a path through the control-flow graph to fill the instruction window, using branch prediction and branch target buffers. *Data-dependence speculation* (or memory dependence speculation) issues loads and stores out of order, speculating that they will access disjoint addresses [82, 139, 146,265].[1] Additionally, *data-value speculation* was proposed to enable early execution of different types of instructions by predicting the result of loads or memory access addresses before they are resolved [236], but we are not aware of any commercial implementations. In all cases, the reorder buffer and store buffer provide version management to recover from mispeculations.

Limitations: Unfortunately, dynamic ILP is nearing, or has reached, its limit [279]; most modern processors execute only one to four instructions in parallel per cycle on average. Superscalar cores are built atop unscalable broadcast networks, associative searches, and centralized structures [359]. The back-end structures, including instruction scheduling logic and register file, grow super-linearly with the instruction window size and number of execution units [210, 278, 342]. Such increases are unacceptable with constrained power and area budgets. Moreover, fundamentally, the amount of parallelism extracted from the sequential interface is bottlenecked by the control-dependent chain of instructions provided to the front-end. It is unlikely that superscalar processors will ever resolve enough branches per cycle to issue thousands of operations in parallel [179, 225, 292, 388]. The instruction-level sequential execution model appears to be insufficiently expressive for a scalable hardware implementation.

By contrast, the Swarm architecture (Chapter 3) performs control and data dependence speculation on ordered tasks of a few tens to hundreds of instructions. Speculating at the granularity of tasks amortizes the costs of task management and speculation. With its execution model of timestamp-ordered tasks, Swarm tasks can create new tasks in a different order than they will commit, conveying new work to hardware as soon as it is available, and enabling highly parallel expressions of work. For example, rather than serially creating tasks, in some cases software constructs a balanced tree of ordered tasks that quickly fills the system with work. The Swarm microarchitecture uses decentralized speculation structures and protocols, enabling it to scale to large system sizes.

2.4.2 Thread-Level Speculation

Thread-level speculation (TLS) is a set of compiler and architectural techniques to automatically extract task-level parallelism from an irregular sequential program, while retaining its original semantics. These ideas developed during the 1990s as superscalar instruction windows were nearing their limits to exploit instruction-level parallelism [24, 225], and research interests turned to extracting thread-level parallelism on a single chip using multiple cores [278] or hardware multithreading [372]. Given the simplicity and widespread adoption of sequential programming, the goal was to find a new source of ILP by creating a larger effective instruction window by speculating at the granularity of tasks [138, 140]. Beyond the summary that follows, the reader can find further detail in an overview by Torrellas [368] and a survey by Estebanez et

[1] At the granularity of tasks, this is the type of speculation typically referenced throughout this thesis.

al. [128].

In the TLS execution model, a program's totally ordered sequence of instructions is split into a totally ordered sequence of tasks. Every task is ordered according to its immediate predecessor and successor tasks to match sequential semantics. TLS compilers [238,297,370,381,404], binary instrumentation [214], or even hardware [13,244] select task boundaries, typically at the start of loop iterations and before and after function calls. Tasks are spawned to express potential irregular parallelism by overlapping task bodies, such as having each task spawn its successor as early as possible [238,347], using a central hardware structure to dispatch tasks to cores [168,344], or recursively spawning a task and its continuation to expose nested parallelism [238,309].

TLS systems [168,214,308,309,344,348,369] extract irregular parallelism by optimistically running the tasks in parallel on distinct cores or hardware threads, speculating that this will not violate sequential semantics. The system tracks the reads and writes of speculatively executing tasks to detect any conflicts that arise when read-after-write (RAW), write-after-read (WAR), or write-after-write (WAW) dependences do not flow in order. If a conflict is detected, or a task was incorrectly speculatively spawned, the system aborts the later-ordered task. The earliest-ordered active task in the system runs non-speculatively, updating main memory directly, and later-ordered speculative tasks that finished running must wait to commit in order. Although software TLS implementations have been proposed [306], the instrumentation overheads are non-negligible for short tasks with low compute-to-memory ratios. We therefore focus on hardware implementations. Like any system that extracts speculative parallelism, TLS speculation is implemented through three key mechanisms: version management, conflict detection, and conflict resolution.

Version management enables recovery from *mispeculation* by separating the non-speculative version of data from the speculatively written versions that could be incorrect. Versioning can be categorized as *lazy*, privately buffering each task's speculative writes until it commits; or *eager*, where each task's writes are made directly to memory, and old values are saved in an undo log. Lazy versioning makes aborts fast, as the buffered state is simply discarded, but commits are slow since updates must be merged into main memory [56,150]. By contrast, eager versioning makes commits fast, as all speculative writes are already in place, but aborts are slow, requiring an undo-log walk to restore the original values to memory. Lazy versioning combined with in-order commits also avoids aborting tasks due to reordered WAR and WAW dependences, as the buffered writes of a later-ordered task do not interfere with its predecessors. Most TLS systems employ lazy versioning to limit the cost of aborts.

The earliest design, Multiscalar [344], uses a centralized buffer that holds multiple speculative versions (*multiversioned*) [139], while later systems decentralize the buffering, for instance in private caches [158,214,348]. A buffer might fill up, requiring hardware mechanisms that carefully overflow speculative state to memory [294], or otherwise stalling the running task. Eager versioning was also explored via software logging [151] and for distributed shared-memory [406]. Garzarán et al. provide a detailed taxonomy of version management in TLS [150].

Conflict detection identifies dependence order violations among running and finished tasks. It is often categorized according to when these checks occur: systems with *eager* conflict detection check for a violation at the time of each data access, whereas *lazy* conflict detection checks after a task has finished, but before it commits. Most TLS systems detect conflicts eagerly to facilitate *speculative data forwarding* which lets tasks access data written by earlier, uncommitted tasks. For instance, a read of an address, a, from a task, T, should find the speculative version written by the latest-ordered task that precedes T, rather than a stale non-speculative version from memory. Speculative forwarding improves performance for ordered tasks with occasional dependences [291, 349].

Conceptually, conflict detection tracks a read and write set of every task's memory accesses as it runs. Practically, these sets are often implemented with speculative read and write bits for every line or word in the private cache [168, 214], or by associating an order tag with every speculative version of a line [139, 344, 348]. RAW order violations are caught when a task writes, for instance by encoding task order information into the invalidations of the coherence protocol [348] or by checking a dedicated disambiguation structure [139, 214] to find later-ordered readers of the data.

Conflict resolution determines what corrective actions must be taken upon a detected conflict. The later-ordered reader may[2] have read stale data and must be aborted and restarted. However, as a result of speculative data forwarding, this aborting task may have forwarded its own speculative writes to later-ordered readers, which in turn may have forwarded their writes, etc. All of these data-dependent tasks must be aborted. Ideally this *cascade* of aborts would be *selective* to avoid wasting useful speculative work: abort only the instigating violator and, recursively, its progeny of spawned tasks and data-dependent successors. As far as we are aware, this strategy has been limited to software TLS schemes [147, 367]. All hardware TLS schemes opt for the simpler implementation that *unselectively* aborts the instigating violator and all later tasks en masse.

Limitations: Although TLS finds speculative parallelism among ordered tasks, it has four key shortcomings spanning the execution model to microarchitecture that curtail its ability to parallelize general fine-grain ordered irregular algorithms at large scale. We address these limitations in this thesis.

First, the purely sequential execution model is well matched to bounded loops but impedes parallelization of irregular algorithms with dynamic task creation, particularly those with priority ordered scheduling. Written sequentially, these algorithms often consist of loops with unknown bounds, where each iteration must push and pop new tasks onto a memory-backed scheduling structure. Because TLS is agnostic to

[2] Value-based conflict detection avoids aborting RAW conflicts if the value is unchanged [244], but requires tracking more state.

task scheduling, *dependences on the scheduling structure*, not necessarily core algorithmic structures, force a chain of data-dependent tasks that cripple otherwise available parallelism (Section 3.1).

Second, while TLS speculation techniques are effective at the scales evaluated (typically fewer than 10 cores), two in particular bottleneck the system as it scales to tens or hundreds of cores: unselective aborts and limited commit throughput. Unselective aborts simplify a TLS microarchitecture, but substantially increase the wasted work incurred by a single conflict as the number of speculative tasks in the system grows. Tasks must commit in order to ensure they have consumed and produced valid data before releasing speculative state. Because TLS systems encode task order through a chain of predecessor-successor pairs [309], they use protocols that *serializes commits* (e.g., passing a token from predecessor to successor [294, 308, 348]). This becomes a bottleneck on large-scale systems with hundreds of fine-grain tasks.

The Swarm architecture (Chapter 3) efficiently expresses and extracts ordered irregular parallelism. Swarm programs consist of dynamic, programmer-controlled timestamp-ordered tasks, and Swarm hardware uncovers parallelism by speculatively executing available tasks out of order. Swarm hardware uses eager versioning with new conflict detection mechanisms that speculatively forward data, while implementing selective aborts to reduce wasted work. Encoding order through integer timestamps confers two key benefits: *(i)* the programmer conveys priority scheduling requirements directly to hardware and thus obviates the false dependences of a memory-backed scheduler, and *(ii)* integer timestamp comparisons enable Swarm to adapt distributed systems techniques to achieve high-throughput ordered commits, avoiding the serializing successor chains of TLS.

Third, TLS schemes dispatch tasks to threads with no concern for data locality. Even at small scales, cache misses hinder TLS scalability [145]. In large-scale systems with hundreds of cores, it is critical to reduce data movement by using caches effectively and running tasks close to the data they access. Spatial hints (Chapter 4) extend the Swarm execution model and exploit Swarm's support for short ordered tasks to enable programmers to exploit locality by sending tasks likely to access the same data to the same tile.

Finally, TLS cannot exploit non-speculative parallelism: only the earliest active task runs non-speculatively. While speculation is an effective approach to extract challenging types of parallelism, some work should be run non-speculatively, such as system services like file or network I/O. Espresso and Capsules (Chapter 5) further extend the Swarm execution model and microarchitecture to reap the efficiency benefits of non-speculative parallelism when it is plentiful, but otherwise leverage speculative execution to scale.

2.4.3 Transactional Memory

Transactional memory (TM) provides optimistic synchronization in explicitly multithreaded programs. Like TLS, the concepts originate from the 1990s and earlier [182, 212, 239, 333], but whereas TLS is rooted in the sequential execution model, TM is based in the multithreaded execution model, seeking improved scalability and ease-of-use [284, 316] compared to blocking approaches like locks. Although the earliest TM proposals [182,212,239] were contemporary with the earliest TLS [138,344], most of the work in TM was done after the surge of interest in TLS dampened. Consequently, in designing Swarm (Chapter 3), we adapt several innovations from TM to the ordered context. Beyond the following summary, more detail can be found in overviews by Harris et al. [171], Guerraoui and Kapałka [162], and Herlihy [181].

In the TM model, a transaction is a sequential block of code executed by a single thread, whose boundaries are annotated by the programmer. Transactions will execute *atomically* and in *isolation*, providing the illusion that every transaction executes all of its operations instantaneously at a single and unique point in time, with no interleaving from other threads. Whereas users of TLS implicitly declare which *single* (sequential) order of tasks is correct, users of TM explicitly declare which unordered blocks of code require atomicity and isolation, so that they appear to execute in *any* serializable order. Transactions may replace locks for critical sections [299,300], while taking care to preserve application semantics [53], or to implement multi-object atomic updates [351] in contrast to single-word atomic instructions like compare-and-swap.

TM systems [69,169,182,262] extract irregular parallelism by running transactions concurrently, speculating that the interleaving of their memory accesses is equivalent to some serializable order. In most TM parlance, a conflict occurs when two or more concurrent transactions access the same datum and at least one access is a write.[3] Transactions can commit concurrently when they do not conflict [56]. Because software TM implementations [333] add non-negligible instrumentation overheads [67, 115], in our pursuit of fine-grain irregular parallelism, we summarize pertinent hardware TM (HTM) implementations. As with TLS, the key mechanisms of TM speculation are version management, conflict detection, and conflict resolution, but the scheduling flexibility of unordered transactions presents new tradeoffs and inspired new designs.

Version management and conflict detection: Whereas most TLS systems combine lazy version management with eager conflict detection, the ability to determine transaction order at run time opens the door to more varied designs in the HTM space. Lazy version management has been combined with lazy conflict detection (e.g., TCC [169] and BulkTM [69]), eager conflict detection (e.g., LTM [17] and VTM [301]), or both configured in software (e.g., FlexTM [336] and Blue Gene/Q HTM [389]). Other designs have used eager version management and eager conflict detection (e.g., Herlihy and Moss [182], UTM [17] and LogTM [262]). As mentioned in Section 2.4.2,

[3] This definition of conflict captures RAW, WAR, and WAW dependences of concurrent transactions.

there are advantages and disadvantages to each design point. Eager versioning makes commits fast but aborts slow, and lazy versioning provides the opposite characteristics. Eager conflict detection reduces the amount of wasted work when an abort is unavoidable. Lazy conflict detection reduces the frequency of communication for conflict checks, and can avoid some aborts by privatizing updates. Bobba et al. conclude that strategic conflict avoidance and resolution policies can address pathologies specific to each design point, resulting in similar performance [56].

Beyond exploring the high-level design points, HTM proposals also innovate on low-level implementations of versioning and conflict detection. Like TLS, some early HTM variants build on existing hardware structures to implement write buffering and read/write-set tracking. For example, they would add per-line speculative access bits to caches [17,169,301] or processor-local buffers [182,299,351], and leverage coherence protocol requests to detect conflicts in a thread's local read or write set. In contrast, later designs sought to decouple speculation mechanisms from these hardware structures and protocols to reduce implementation complexity or enable transactions of unbounded size. LogTM [262] decouples version management from the caches, moving to memory-resident undo logs, while still tracking read and write sets with per-line bits in the cache. To support unbounded size, when a transaction overflows the L1 cache, LogTM sets a *sticky* coherence state in the directory so that later potentially conflicting requests are forwarded to the overflower. The Bulk TM and TLS system [69] decouples conflict detection from the caches and coherence protocol. It represents read and write sets compactly in Bloom filter [50] *signatures* and broadcasts signatures to detect conflicts at commit time. LogTM-SE [399] combines these two ideas: it eagerly detects conflicts on coherence requests, tracks address read/write sets in signatures, and implements eager versioning with memory-resident undo logs.

Conflict resolution and transaction ordering: Unordered transactions grant flexibility in how to resolve a conflict to preserve some serializable execution. Early HTM systems use simple policies that stall or abort some transaction(s) upon a conflict. For example, some lazy-lazy systems have the *committer win* conflicts, thereby aborting all other conflicting transactions [69]. More recent systems have targeted *conflict serializability* to improve performance and efficiency by reducing the frequency of aborts and stalls. By applying speculative data forwarding for some dependences, neither conflicting transaction need abort or stall if the transactions are reordered so that the predecessor in the conflicting dependence commits first [25, 199, 296, 302].

While transactions are typically defined to be unordered, some hardware [69, 167, 169, 291] and software [61, 157] TMs let programmers control the commit order among transactions, bridging the gap between TM and TLS. Other HTMs order transactions internally, either to avoid pathologies [56, 262] or to implement conflict serializability [25,137,199,296,302]. However, this order is not controllable by programmers.

Limitations: Although TM systems provide new mechanisms to extract speculative irregular parallelism, including versioning and conflict detection, three main issues prevent it from effectively supporting fine-grain ordered irregular parallelism at large scale.

First, and most clearly, most HTMs do not support ordered tasks. Like TLS, the few HTMs with programmer-controlled commit order, such as TCC [169] and Bulk [69], do not support dynamic task creation and so similarly must resort to memory-resident task queues and suffer their false dependences. By contrast, the Swarm (Chapter 3) execution model consists of dynamic, programmer-controlled timestamp-ordered tasks. This conveys the required commit order directly to hardware, and enables hardware to provide task management, free of any false dependences.

Second, HTMs are agnostic to data locality, so, like TLS, data movement precludes their scalability to hundreds of cores. Prior work has considered *when* to run particular transactions, proposing schedulers that limit concurrency to reduce the abort rate under high contention [19, 44, 402]. However, reducing data movement requires controlling *where* particular transactions should run. We address this problem with spatial hints (Chapter 4), which exploit programmer knowledge to convey to hardware which tasks should run at the same tile to reduce data movement.

Finally, existing HTM techniques to combine speculative and non-speculative parallelism do not work in the context of ordered parallelism. Although TM programs consist of speculative (transactional) and non-speculative (non-transactional) code, it is cumbersome to coordinate accesses to shared data from both types of tasks. Espresso (Chapter 5) extends Swarm and spatial hints to support both speculative and non-speculative ordered tasks, and provides synchronization mechanisms to coordinate shared accesses from both types of tasks. To enable transactions to perform I/O in parallel or avoid conflicts by leveraging application-specific knowledge, prior work in HTM has proposed escape actions [64, 264, 411] and open-nested transactions [251, 264, 266] to selectively bypass hardware speculation. Unfortunately these techniques are unsafe with speculative forwarding, a critical optimization to exploit ordered parallelism, which we address with Capsules in Chapter 5.

CHAPTER 3

Swarm: A Scalable Architecture for Ordered Parallelism

This work was conducted in collaboration with Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. The ideas for the execution model, virtual-time based conflict detection, selective aborts, and distributed commit protocol were developed collaboratively. This thesis contributes task queuing and prioritization, fixed-sized queue management, the software runtime, and the oracle tool. This thesis also contributes to the development of applications, and the architectural simulator.

In this chapter, we focus on unlocking the parallelism in ordered irregular algorithms, or *ordered irregular parallelism*. Ordered parallelism is abundant in many domains, such as simulation, graph analytics, and databases. For example, consider a timing 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 reads and modifies a specific component (possibly 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 on commodity multicores but has found that runtime overheads, including priority ordering and task scheduling, negate the benefits of parallelism [176, 177]. This motivates the need for architectural support.

To guide our design, we first characterize several applications with ordered irregular parallelism (Section 3.1). We find that tasks in these applications are as small as a few tens of instructions, and, unlike unordered tasks, often cannot be coarsened into larger code blocks due to the order constraints. Moreover, many of these algorithms rarely have true data dependences among tasks, and their maximum achievable parallelism exceeds 100×. It may seem that thread-level speculation (TLS), which speculatively parallelizes sequential programs (Section 2.4.2), could extract this type of parallelism. However, that this is not the case due to two reasons (Section 3.1.3):

- Ordered irregular 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, sequential implementations of timing simulators use a priority queue to hold future tasks. Priority queue accesses introduce false data dependences that limit the effectiveness of TLS.
- To scale, ordered irregular algorithms need very large speculation windows, of thousands of tasks (hundreds of thousands of instructions). Prior TLS schemes use techniques that scale poorly beyond few cores and cannot support large speculation windows.

This chapter presents Swarm, an architecture that tackles these challenges. Swarm consists of *(i)* a task-based execution model that conveys task order constraints directly to hardware to sidestep false data dependences (Section 3.2), and *(ii)* a supporting microarchitecture that leverages this execution model to scale efficiently (Section 3.3).

Swarm is a tiled multicore with distributed task queues, speculative out-of-order task execution, and ordered task commits. Swarm adapts prior eager version management and conflict detection schemes [262, 399], and features several new techniques that allow it to scale. Specifically, we make the following novel contributions:

- An execution model based on tasks with programmer-specified timestamps that conveys order constraints to hardware without undue false data dependences on a software scheduler.
- A hardware task management scheme that features speculative task creation and dispatch, drastically reducing task management overheads, and 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 (unlike prior TLS schemes that forward speculative data, which unselectively abort all later tasks).
- A distributed commit protocol that allows ordered commits without serialization, supporting multiple commits per cycle with modest communication (unlike prior schemes that rely on successor lists, token-passing, and serialized commits).

We evaluate Swarm in simulation (Section 3.4 and Section 3.5) using six challenging workloads: four graph analytics algorithms, a discrete-event simulator, and an in-memory database. At 64 cores, Swarm achieves speedups of 51–122× over a single-core Swarm system, and outperforms state-of-the-art parallel implementations of these algorithms by 2.7–18.2×. 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.

3.1 Motivation

3.1.1 Understanding Ordered Irregular Parallelism

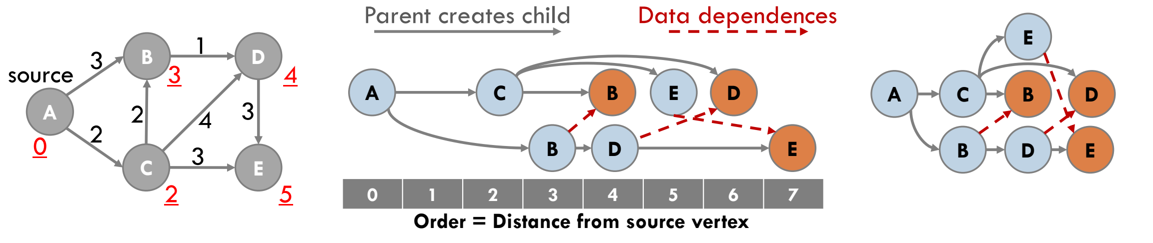
Ordered irregular algorithms have three main characteristics [176, 289], as discussed in Section 2.1. First, they consist of tasks that must follow a programmer-defined total or partial priority order. Second, not all tasks are known in advance. Instead, tasks dynamically create children tasks and schedule them to run at particular future times, potentially resulting in different task creation and execution orders. Third, tasks may have data dependences that are not known a priori. Data must flow from a predecessor task to its successor task(s), according to the order constraints.

These algorithms are common in many application domains. First, they are common in graph analytics, especially in search problems [111, 173] but also in vertex- or edge-set selection problems [216, 391]. Second, they are important in simulating systems whose state evolves over time. These include circuits [259], computers [75, 307], networks [197, 378, 379], healthcare systems [207], and systems of partial differential equations [175, 234]. Third, they are needed in systems that must maintain externally-imposed order constraints. Consider geo-replicated databases where transactions must appear to execute in timestamp order [92], or deterministic runtimes [237], deterministic architectures [106] and record-and-replay systems [191, 395] that constrain the schedule of parallel programs to ensure deterministic execution. Finally, even applications with large unordered tasks could have more parallelism if expressed as short ordered tasks. For example, an in-memory database in high contention has little parallelism among its unordered transactions. Instead we can view a database transaction's underlying queries and updates as short, ordered tasks, while retaining the apparent atomicity by ordering the tasks of one transaction before those of another.

Dijkstra's algorithm [111, 141] for the single-source shortest paths (`sssp`) problem aptly illustrates the challenges in expressing the parallelism of an ordered algorithm. `sssp` finds the shortest distance between some source vertex and all other vertices in a graph with non-negative weighted edges. Listing 3.1 shows the sequential code for `sssp`, which uses a priority queue to schedule tasks. Each task, dequeued from the queue, visits a single vertex, and is ordered by its projected distance to the source vertex. `sssp` relies on task order to guarantee that the first task to visit each vertex comes from a shortest path. This task sets the vertex's distance and enqueues children

```
prioQueue.enqueue({0, source});
while (!prioQueue.empty()) {
  (distance, v) = prioQueue.dequeueMin();
  if (v->distance == UNSET) {
    v->distance = distance
    for (Vertex* n : v->neighbors) {
      int projected = distance + length(v, n);
      prioQueue.enqueue({projected, n});
    }
  } else { /* vertex already visited */ }
}
```

Listing 3.1: Dijkstra's single-source shortest paths (sssp) sequential algorithm, highlighting the visited and non-visited paths that each task may follow.



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

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

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

Figure 3-1: Dijkstra's sssp algorithm has plentiful ordered irregular parallelism.

for every neighbor. Later tasks visiting the same node do nothing. Figure 3-1(a) shows an example input graph, and Figure 3-1(b) shows the resulting task graph that sssp executes to process this input. Figure 3-1(b) shows the order of each task (its distance to the source vertex) 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 vertex B, so they have a data dependence.

A distinctive feature of many ordered algorithms is that task creation and execution orders can be 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(b), $(C, 2)$ creates $(B, 4)$, but running $(B, 4)$ immediately would produce the wrong result, because $(B, 3)$, created by a different parent, must run first. This feature necessitates dynamic task scheduling. Sequential implementations of these programs use scheduling data structures, such as priority or FIFO queues, to process tasks in the right order. This is a key reason why TLS cannot exploit ordered parallelism in general: the in-memory scheduling data structures introduce false data dependences among otherwise independent tasks (Section 3.1.3). In contrast, the Swarm execution model (Section 3.2) dispenses with software scheduling, conveying the task order constraints directly to hardware.

Order constraints limit non-speculative parallelism. For example, `sssp` admits a one-distance-at-a-time parallelization [107, 230]. At any given time, only tasks with the lowest unprocessed distance are executed in parallel; these create tasks with higher distances. Exploiting the unordered parallelism within each distance works well on a shallow graph with many vertices that have the same distance to the source. However, weighted graphs (e.g., road maps) often have very few vertices per distance, so there is little work to do at each step, and limited unordered parallelism to be found [177]. For example, in Figure 3-1(b), this strategy permits only $(B, 4)$ and $(D, 4)$ to run in parallel.

To extract the full amount of latent ordered parallelism, we must run independent tasks out of order. For example, Figure 3-1(d) shows an ideal schedule for the `sssp` tasks. It processes independent tasks across multiple distances simultaneously, shown at each x-axis position. This schedule achieves twice the parallelism of a serial schedule on this small graph, and larger graphs permit even more parallelism (Section 3.1.2). The illustrated schedule produces the correct result because, although it elides order constraints among independent tasks, it ensures that data dependences flow in the right order. Unfortunately, the tasks and their data dependences are not known in advance, so only an oracle could devise this schedule. Therefore, to elide unnecessary order constraints at run time, we must resort to speculative execution. Specifically, for every task other than the earliest active task, Swarm speculates that there are no data-dependent predecessor tasks, and it executes the task anyway (Section 3.3). If this guess is wrong, Swarm detects dependence order violations and aborts offending tasks to preserve correctness.

3.1.2 Analysis of Ordered Irregular Algorithms

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

Benchmarks: We analyze six 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 [173] to find the shortest route between two points in a road map.
- **`msf`** is Kruskal's minimum spanning forest algorithm [93, 216].
- **`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 [371].

Section 3.4 describes their input sets and methodology details, including task selection.

Oracle analysis tool: We developed a pintool [242] 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.

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×
Instructions	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 irregular applications.

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 [396] (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: Speculative parallelism is plentiful. These applications have at least 158× maximum parallelism (`msf`), and up to 3440× (`bfs`). Thus, most order constraints are superfluous, making *speculative execution* attractive.

Insight 2: Tasks are small. Across the benchmark suite, 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, as is done with unordered tasks [176, 252]. *Hardware support for task management* drastically reduces these overheads.

Insight 3: A large speculation window is needed. Table 3.1 also shows the achievable parallelism within a limited task window. When finding the shortest schedule 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 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 that an oracle can see, while incurring only moderate overheads.

3.1.3 Limitations of Thread-Level Speculation

As discussed in Section 2.4.2, prior work has investigated thread-level speculation (TLS) schemes to parallelize sequential programs [150, 168, 309, 344, 349]. They ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Although TLS schemes support speculatively synchronized ordered tasks, we find that two critical problems prevent them from exploiting ordered irregular parallelism in general.

The TLS execution model limits parallelism: To be parallelized by TLS, ordered algorithms must be expressed as sequential programs, but for those with dynamically scheduled tasks, their sequential implementations limit parallelism. Consider the `sssp` code in Listing 3.1, where each iteration dequeues a task from the priority queue and runs it, potentially enqueuing more tasks. Data dependences *in the scheduling queue*, not among tasks themselves, cause frequent conflicts and aborts. For example, iterations that enqueue high-priority (low-distance) 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 do not need to queue dynamic tasks: their task orders match loop iteration order.

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

TLS scalability bottlenecks: Although prior work has developed scalable versioning and conflict detection schemes [70, 295, 348], two challenges limit TLS performance with large speculation windows and small tasks: unselective aborts and limited commit throughput.

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 [291, 349]. Speculative data forwarding 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 en masse* [150, 168, 309, 344, 348]. TCC [169] and Bulk [69] are the exception: they do not forward data and only abort later readers when the earlier writer commits.

We similarly find that forwarding speculative data is crucial for Swarm. However, although it is reasonable to abort all later tasks with small speculative windows (2 to 16 tasks are typical in prior work), Swarm has a 1024-task window, making unselective aborts impractical. To address this, our novel conflict-detection scheme forwards speculative data and selectively aborts only dependent tasks upon mispeculation.

Commit serialization: Prior TLS schemes enforce in-order commits by passing a token among ready-to-commit tasks [168, 309, 344, 348]. 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 64-cycle tasks, a 64-core system should commit 1 task/cycle on average. Even with constant-time [294] or instantaneous commits, the latency incurred by passing the token makes this throughput unachievable.

Instead, we adapt techniques from distributed systems to achieve in-order commits without serialization, token-passing, or building successor lists.

3.2 Swarm Execution Model

Swarm is a co-designed task-based execution model and hardware microarchitecture with ordered tasks at the interface. Swarm programs consist of dynamically created *timestamp*-ordered tasks that can read and write arbitrary data in shared memory. The program's output will always match that of a sequential model where task execution is scheduled by a monotone [365] priority queue. Every task's timestamp acts as its key in the modeled queue. Swarm guarantees that tasks appear to run in increasing timestamp order, as if a single thread repeatedly dequeues the lowest-timestamp task from the queue, runs it, then dequeues the next task, until the queue is empty. Any task can create children tasks, enqueuing them into the modeled priority queue with timestamps greater than or equal to the parent's timestamp.[1] Swarm retains atomicity with partial orders: tasks with equal timestamp are ordered arbitrarily among themselves, but they appear to run in some serial order, while ensuring every child is ordered after its parent. To simplify the discussion in this chapter, we defer the definition and implementation of the exception model, including handling system calls, to Chapter 5.

[1] Among other semantics, Fractal (Chapter 6) can generalize the Swarm execution model to a non-monotone priority queue: any child with lower timestamp than its parent can be enqueued to the parent's *subdomain*.

Programs leverage the Swarm execution model through a simple API. Tasks create children tasks by calling the following inlined, non-blocking function:

```
swarm::enqueue(taskFn, timestamp, args...)
```

The new task, when dispatched, will run a function with the following signature, with arguments supplied through registers:

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

Performance considerations from software: Swarm's timestamp-ordered execution model decouples task creation and execution orders: software can convey new work to hardware as soon as it is discovered, rather than in the order it needs to run. This exposes a large amount of parallelism that, as we detail in Section 3.3, hardware extracts by speculatively running tasks out of order. Swarm hardware implements the centralized priority queue abstraction through scalable distributed structures.

Example: Listing 3.2 illustrates the execution model through the Swarm implementation of Dijkstra's `sssp`. The code closely resembles the sequential implementation from Listing 3.1—there is no explicit synchronization or thread management. Listing 3.2 defines a single task function, `ssspTask`. Because every `sssp` task is ordered according to its vertex's projected path distance to the source, that distance is used directly as a timestamp. If its own vertex `v` is unvisited, `ssspTask` creates one child task for each neighbor, with the neighbor's projected distance as the timestamp. Because tasks appear to execute in timestamp order, the first task to visit each vertex will come from a shortest path. A program invokes Swarm by enqueuing some initial tasks with `swarm::enqueue` and calling `swarm::run`, which returns control when all tasks finish. Listing 3.2 creates one initial task to visit the source vertex with distance zero to itself, then initiates speculative execution.

To reduce the total number of executed tasks, and consequently the pressure on hardware resources, we can also restructure the task code into a slightly coarser granu-

```
void ssspTask(Timestamp distance, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = distance;
    for (Vertex* n : v->neighbors) {
      Timestamp projected = distance + length(v,n);
      swarm::enqueue(ssspTask, projected, n);
    }
  } else { /* vertex already visited */ }
}

void main() {
  // initialization ...
  swarm::enqueue(ssspTask, 0, source);
  swarm::run();
}
```

Listing 3.2: Swarm implementation of Dijkstra's `sssp` algorithm. The code is similar to the sequential implementation (Listing 3.1). Synchronization is implicit through task order.

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

Listing 3.3: Swarm implementation of Dijkstra's `sssp` using fewer larger tasks.

larity, shown in Listing 3.3. This variant is evaluated in this chapter. Chapter 4 explores the trade-off between memory footprint, number of tasks, and locality in detail.

3.3 Swarm Implementation

The Swarm microarchitecture uncovers ordered parallelism by speculatively executing tasks out of order, but enforcing order on data-dependent tasks. It introduces modest changes to a tiled, cache-coherent multicore, shown in Figure 3-2. Each tile has a group of simple single-threaded cores.[2] Each core has small, private, write-through L1 caches. All cores in a tile share an L2 cache, and each tile has a slice of a fully shared NUCA L3 cache. Each tile is augmented with a *task unit* that queues, dispatches, and commits tasks. Tiles communicate through a mesh NoC.

Swarm is carefully designed to support tiny tasks and a large speculation window efficiently. Swarm has no centralized structures: every tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Task units only communicate *(i)* when they send new tasks to each other to maintain load balance, *(ii)* to notify children of parent commit or abort, and, infrequently *(iii)* to determine which finished tasks can be committed.

[2] Our prior work [5] expands Swarm with multithreaded cores.

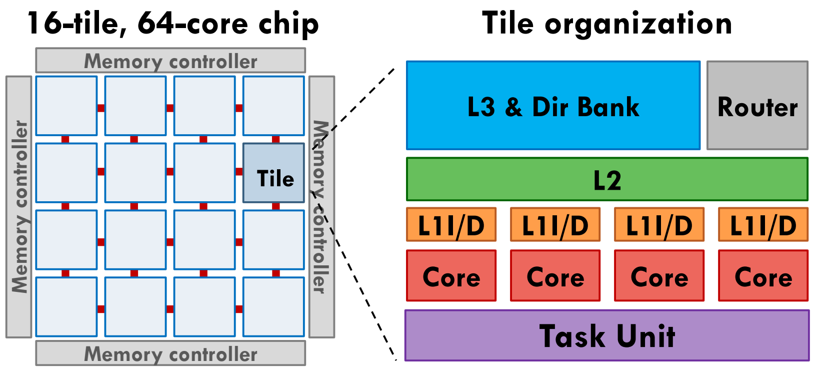
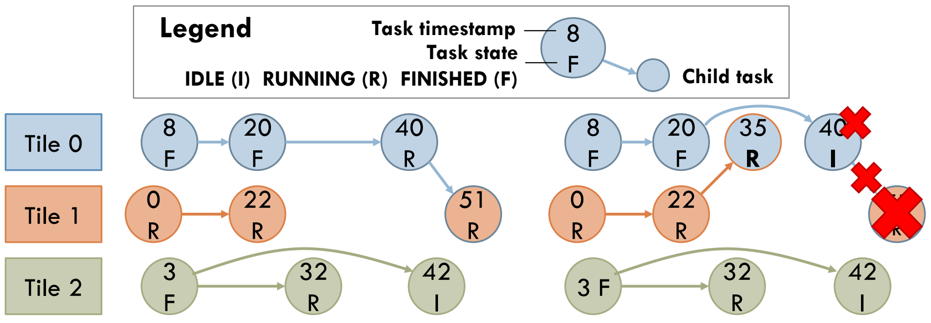


Figure 3-2: Swarm 64-core chip and tile configuration.



(a) Tasks far ahead of the minimum timestamp (0) are run, even when parent is still speculative (b) Selective aborts: Task 35 aborts 40 and child 51, but not 42

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.

Swarm speculates far ahead of the earliest active task, and runs a task even if its 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.

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

3.3.1 ISA Extensions

Swarm manages and dispatches tasks using hardware task queues. A task is represented by a descriptor with the following architectural state: the function pointer, a 64-bit timestamp, and the task's arguments. 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. 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. Because hardware tracks parent-child relations to facilitate rollback (Section 3.3.5) and virtualization (Section 3.3.7), tasks may create a limited number of children (8 in our implementation). Tasks that need more children use a runtime API that creates a balanced tree of recursive *enqueuer* tasks to enqueue the children.

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.

This ISA minimizes task creation and dispatch costs, enabling tiny tasks: a single instruction creates each task, and arguments are copied from/to registers, without stack accesses.

3.3.2 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 queue entry when it finishes execution, and also deallocates it at commit time. For now, assume these queues always have free entries. Section 3.3.7 describes our solutions for 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

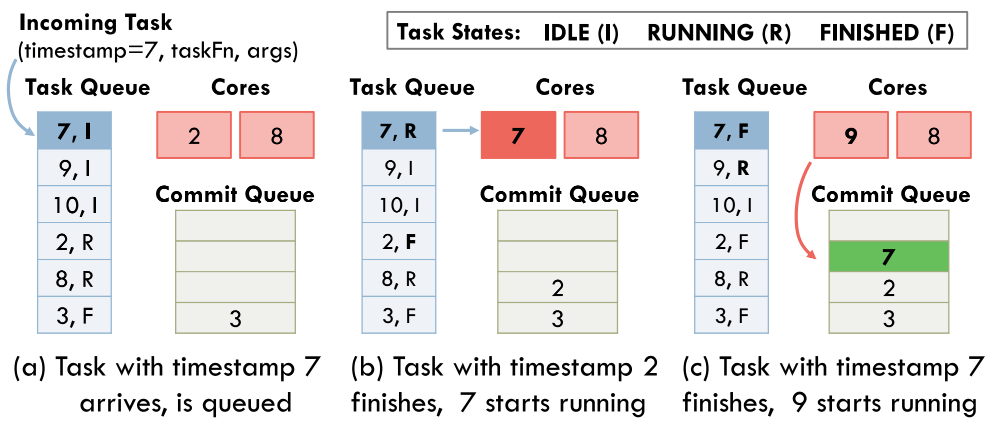

Incoming Task
(timestamp=7, taskFn, args)
Task States: IDLE (I) RUNNING (R) FINISHED (F)
Task Queue
Cores
Commit Queue
7, I
9, I
10, I
2, R
8, R
3, F
2
8
3
7, R
2, F
7
(a) Task with timestamp 7 arrives, is queued
(b) Task with timestamp 2 finishes, 7 starts running
7, F
9, R
9
(c) Task with timestamp 7 finishes, 9 starts running

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

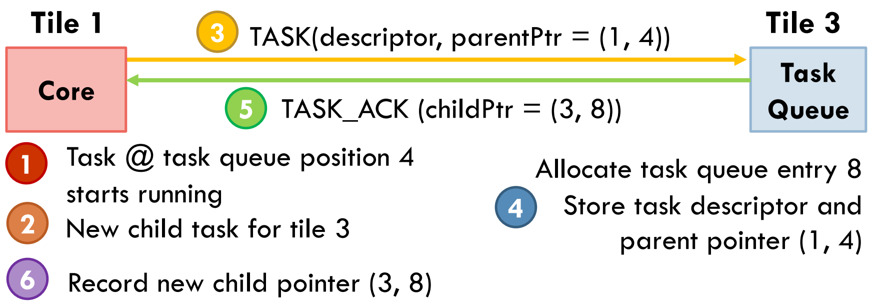

Tile 1
Core
3 TASK(descriptor, parentPtr = (1, 4))
5 TASK_ACK (childPtr = (3, 8))
Tile 3
Task Queue
1 Task @ task queue position 4 starts running
2 New child task for tile 3
6 Record new child pointer (3, 8)
4 Allocate task queue entry 8 Store task descriptor and parent pointer (1, 4)

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.

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 (timestamp) 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`), its task unit sends a request to enqueue the task to a tile following the protocol in Figure 3-5. The instruction is non-blocking, so the latency of remote enqueue is off the core's critical path. To balance loads across tiles, Swarm selects a random destination tile. Chapter 4 considers how to choose the destination to improve data locality. 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. Until a `TASK_ACK` is received, the child task descriptor occupies an entry in the parent's local task queue in an unqueued state; it is only eligible to be executed at its target tile.

Task dispatch prioritization: Tasks are prioritized for execution in increasing timestamp order. When a core issues `dequeue_task`, the highest-priority local idle task is selected and dispatched 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 [283], 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 3.3.8).

3.3.3 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,

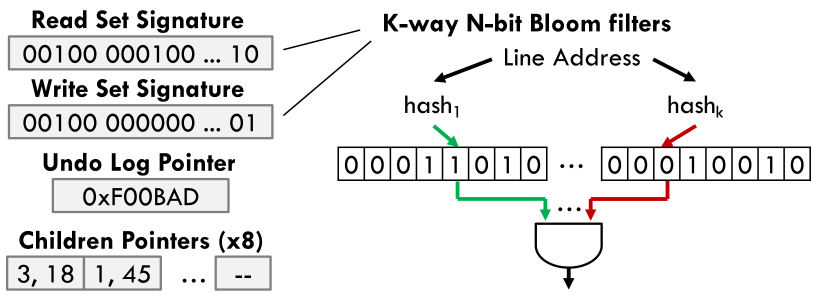


Figure 3-6: Speculative state for each task. Each core and commit queue entry maintains this state. Read and write sets are implemented with space-efficient Bloom filters.

but aborts are slow. However, Swarm's execution model makes conflicts rare, so eager versioning provides the right tradeoff.

Eager versioning is common in hardware transactional memory (HTM) systems [262, 399], most of which do not perform ordered execution or speculative data forwarding (Section 2.4.3). By contrast, most TLS systems use lazy versioning (buffering speculative data in caches) or more expensive multiversioning [69,150,168,169,291,308,309, 344, 348, 349] to limit the cost of aborts (Section 2.4.2). Some TLS schemes are eager [150,151,406], and they still suffer from the limitations described in Section 3.1.3.

Swarm's speculative execution borrows from LogTM and LogTM-SE [262,325,399]. Our key contributions over these and other speculation schemes are *(i)* conflict detection (Section 3.3.4) and selective abort techniques (Section 3.3.5) that leverage Swarm's hierarchical memory system and Bloom filter [50] signatures to scale to large speculative windows, and *(ii)* a technique that exploits Swarm's large commit queues to achieve high-throughput commits (Section 3.3.6).

Figure 3-6 shows the per-task state needed to support speculation: read- and write-set signatures, a pointer to a heap-allocated undo log, and child task pointers. Every 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 3.3.4). Then, it inserts the addresses read and written into the Bloom filters, and, for every write, it saves the previous memory value in a memory-resident undo log. Stack addresses are neither conflict-checked nor 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 to execute another task.

3.3.4 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: Some tasks may have equal programmer-assigned timestamps. 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.

This greedy dequeue-time assignment achieves a simple implementation of virtual time. However, for unordered applications or very wide partial orders, this may cause unnecessary aborts among same-timestamp tasks. In prior work [5] we describe a scheme that assigns virtual time lazily upon a task's first conflict, borrowing ideas from conflict serializable HTMs [199], trading some complexity for some performance improvement in unordered applications.

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 [150].

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 [262], 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

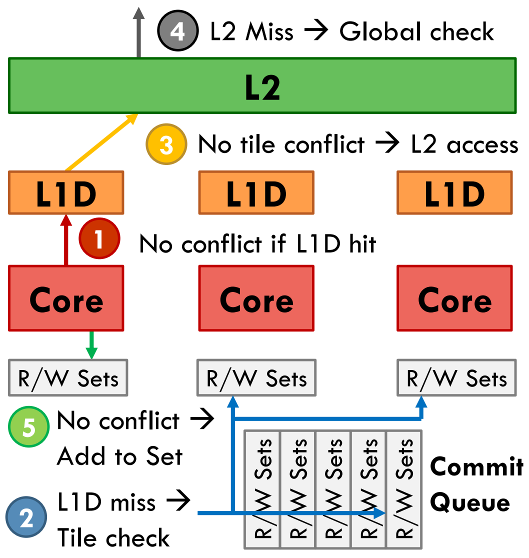


Figure 3-7: Local, tile, and global conflict detection for an access that misses in the L1 and L2.

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.

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.

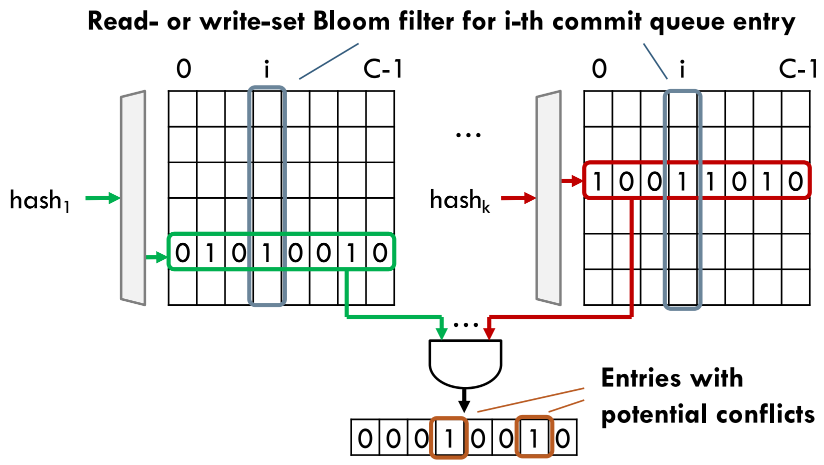

Read- or write-set Bloom filter for i-th commit queue entry
0
i
C-1
hash$_1$
0 1 0 1 0 0 1 0
hash$_k$
1 0 0 1 1 0 1 0
Entries with potential conflicts
0 0 0 1 0 0 1 0

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.

3.3.5 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 [85, 296], would require $1024 \times 1023 \simeq 1$ Mbit of state.

3.3.6 Scalable Ordered Commits

To achieve high-throughput commits, Swarm adapts the virtual time algorithm [200], common in parallel discrete event simulation [134]. 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 *(time-stamp, current cycle, tile id)* for the purposes of this algorithm. The arbiter computes

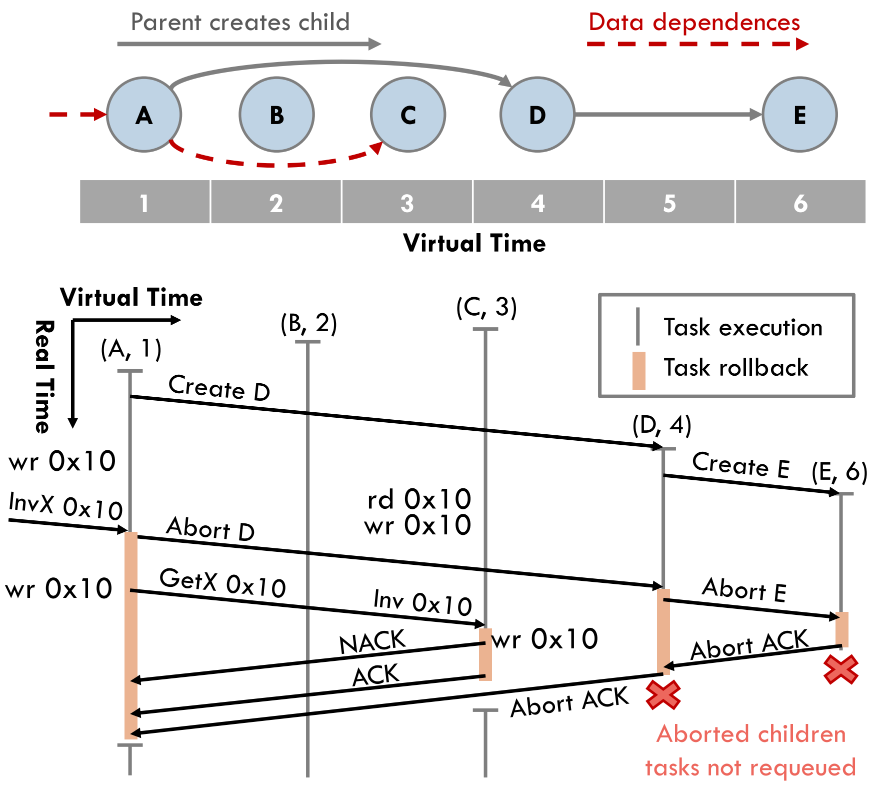
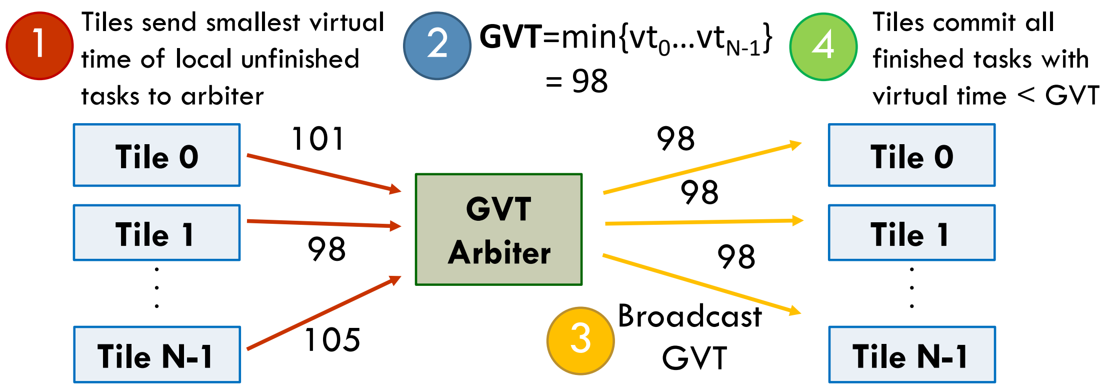


Figure 3-9: Selective abort protocol. Suppose $(A, 1)$ must abort after it writes 0x10, due to an incoming invalidation from some other tile. $(A, 1)$'s abort squashes child $(D, 4)$ and grandchild $(E, 5)$. During rollback, A also aborts $(C, 3)$, which read A's speculative write to 0x10. $(B, 2)$ is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



1 Tiles send smallest virtual time of local unfinished tasks to arbiter
2 GVT=min{vt0...vtN-1} = 98
4 Tiles commit all finished tasks with virtual time < GVT
Tile 0
Tile 1
Tile N-1
101
98
105
GVT Arbiter
98
98
98
3 Broadcast GVT
Tile 0
Tile 1
Tile N-1

Figure 3-10: Global virtual time commit protocol. Tiles periodically communicate with an arbiter to determine the earliest active task in the system. All tasks that precede this earliest active task can safely commit.

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 2.4.2 and Section 3.1.3), this scheme does not even require finding the successor and predecessor of each task, and does not serialize commits.

For the system sizes evaluated in this chapter, we find that a single GVT arbiter suffices. For larger systems in later chapters, we use a hierarchy of arbiters that form min-reduction and GVT-broadcast trees, to reduce communication.

3.3.7 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 [161, 221, 326]. When a tile's task queue is nearly full, the local task unit dispatches a special, non-speculative *coalescer* task to one of the cores. This coalescer task removes several 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 *spilled* tasks. The coalescer's timestamp matches that of the locally earliest active task to hold the GVT below the spilled tasks. The splitter's timestamp matches that of the earliest spilled task, to deprioritize its dispatch until the tasks are needed.

A task queue entry is only spilled to memory if it is *untied*: the task has no parent or its parent has already committed. This simplifies parent-child abort notification. When every entry is still *tied* to the commit or abort of its parent, we need another approach.

Virtual time-based allocation: The task and commit queues may also fill up with tied tasks. The general rule to avoid deadlock due to resource exhaustion is to always prioritize resources toward coalescer tasks and then 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, the task unit aborts the highest-virtual time finished task and t takes its commit queue entry. Otherwise, t stalls its core, waiting for an entry.

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

Task queue: A fraction of the task queue capacity is reserved for untied tasks only (one third in our implementation). An enqueue request for an untied task is allocated any free entry, while a request for a tied task is restricted to the remaining capacity. Consequently, when the queue is full, some tasks are always eligible to be spilled by a coalescer. Suppose an enqueue request for task t arrives at a task unit but the relevant capacity has been reached. The enqueue request is NACK'd (instead of ACK'd as in Figure 3-5) and the *parent task's task unit* retries the enqueue using linear backoff. If all cores are running non-coalescer tasks, one of these running tasks is aborted so that a coalescer can run to free space for later enqueue requests.

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. The task runs non-speculatively: it need not keep track of its children (no child pointers), so when those children are sent to another tile, they arrived untied, and can be spilled to memory if the task queue is full. This ensures that the GVT task makes progress, avoiding deadlock.

3.3.8 Analysis of Hardware Costs

Figure 3-11 summarizes Swarm's hardware changes. Swarm adds task units, a GVT arbiter (or a hierarchy of them in large systems), and modifies cores and caches.

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. In this chapter 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).[3] We use CACTI [366] for the task and commit queue SRAM areas (using 32 nm ITRS-HP logic) and scaled numbers from a commercial 28 nm TCAM [16] 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

[3] We exploit locality to scale the system to hundreds of cores in Chapter 4.

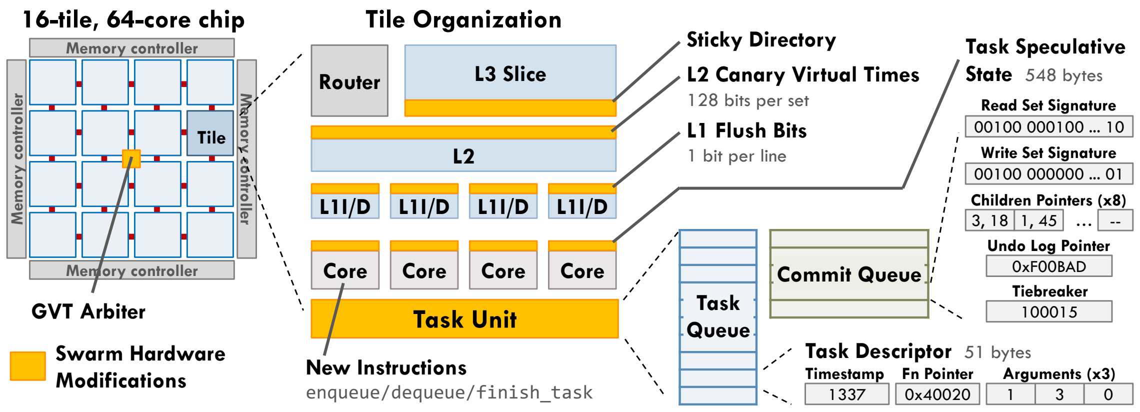


Figure 3-11: Summary of Swarm hardware modifications. Swarm augments a tiled multicore with a GVT arbiter, a task unit on each tile, and modifications to cores and caches. These enhancements require moderate cost.

		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.

~70pJ per access [274]. 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 3.3.1), 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 an in-house microarchitectural, event-driven, sequential simulator based on Pin [242,282] to model Swarm systems of up to 64-cores with a 3-level cache hierarchy. In this chapter we use simple IPC-1 cores with detailed timing models for caches, on-chip network, and main memory (adapted from zsim [323]), and also model Swarm features (e.g., conflict checks, aborts, etc.) in detail. In all following chapters we model up to 256-core systems with in-order cores. Table 3.3 details the

Cores	64 cores in 16 tiles (4 cores/tile), 2 GHz, x86-64 ISA, IPC-1 except misses and Swarm instructions
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	16 MB, shared, static NUCA [211] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories, no silent drops
NoC	4×4 mesh, 256-bit links, X-Y routing, 3 cycles/hop
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (4096 total), 16 commit queue entries/core (1024 total)
Swarm instrs	5 cycles per `enqueue/dequeue/finish_task`
Conflicts	2048-bit 8-way Bloom filters, H_3 hash functions [66] Tile checks take 5 cycles (Bloom filters) + 1 cycle for every timestamp compared in the commit queue
Commits	Tiles and GVT arbiter send updates every 200 cycles
Spills	Coalescers fire when a task queue is 75% full Coalescers spill up to 15 tasks each

Table 3.3: Configuration of the 64-core system.

modeled configuration.

Benchmarks: We use the six benchmarks mentioned in Section 3.1.2: `bfs`, `sssp`, `astar`, `msf`, `des`, and `silo`. Table 3.4 details their provenance and input sets.

For most benchmarks, we use tuned serial and state-of-the-art parallel versions from existing suites (Table 3.4). Like `sssp`, benchmarks `bfs`, `astar`, and `des` dynamically create ordered tasks, so their serial implementations repeatedly dequeue work from a scheduling FIFO or priority queue. In contrast, the key loop of `msf` is bounded, iterating over graph edges in decreasing weight order. We port these serial implementation to Swarm by obviating the queue, and conveying task priority order to hardware through timestamps. Therefore Swarm implementations delineate fine-grain tasks, but otherwise 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.

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 [60]. 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

	Software baselines	Input	Seq run-time
bfs	PBFS [230]	hugetric-00020 [26, 100]	3.68 Bcycles
sssp	Bellman-Ford [176, 289]	East USA roads [1]	4.42 Bcycles
astar	Own	Germany roads [280]	2.08 Bcycles
msf	PBBS [339]	kronecker_logn16 [26, 100]	2.16 Bcycles
des	Chandy-Misra [176, 289]	csaArray32 [289]	3.05 Bcycles
silo	Silo [371]	TPC-C, 4 whs, 32 Ktxns	2.93 Bcycles

Table 3.4: Benchmark information: source of baseline implementations, inputs, and run-time of the serial version.

intra-transaction parallelism. Tasks from different transactions use disjoint timestamp ranges to preserve atomicity. We disable logging to disk. Whereas prior work used TLS to extract intra-transaction parallelism only [87, 88], our approach exposes significant fine-grain 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.

All benchmarks have serial run-times of over two 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 [230]. 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: Only two of the benchmarks used in Section 3.5 (`des` and `silo`) allocate memory within tasks. To separate concerns, dynamic memory allocation is not simulated in detail, until a scalable solution is presented in Chapter 5. As we show in that chapter, data dependences in the system's memory allocator would otherwise serialize all allocating tasks. Instead, in this chapter, Chapter 4, and Chapter 6, 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*.

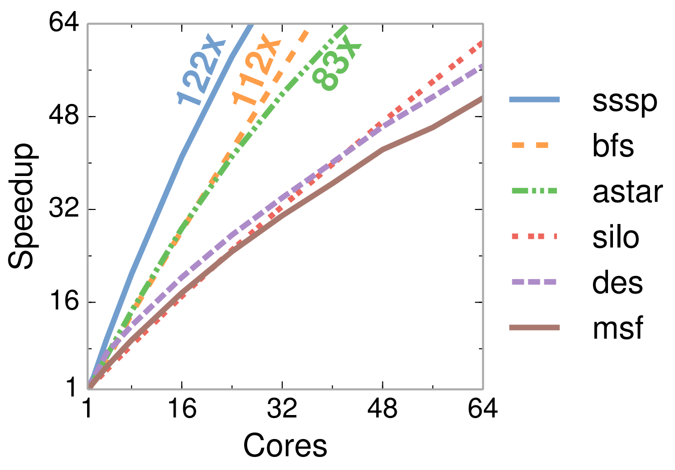


Figure 3-12: Swarm self-relative speedups on 1-64 cores. Larger systems have larger queues and caches, which affect speedups and sometimes cause superlinear scaling.

3.5 Evaluation

We first compare Swarm with alternative implementations, then analyze its behavior in depth.

3.5.1 Swarm Scalability

Figure 3-12 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. *Larger systems have higher queue and cache capacities*, which sometimes causes superlinear speedups.

Each line in Figure 3-12 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× (`msf`) to 122× (`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 3-13 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 3-12, queue and L2/L3 capacities scale with the number of cores.

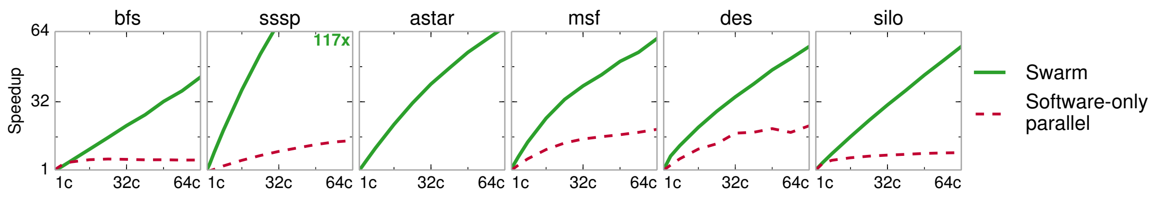


Figure 3-13: Speedup of Swarm and state-of-the-art software-parallel implementations from 1 to 64 cores, relative to a tuned serial implementation running on a system of the same size.

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

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 [230], 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 [93]. 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 [176], 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 [93]. 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 speculation via deterministic reservations [46], 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 [259, 346]. 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

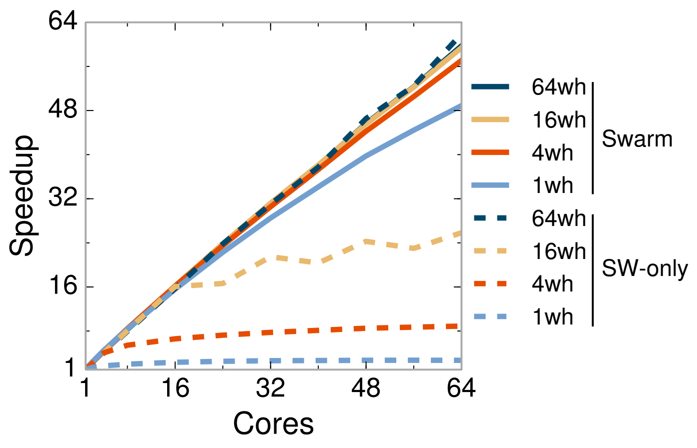


Figure 3-14: Speedup of Swarm and software `silo` with 64, 16, 4, and 1 TPC-C warehouses.

software-parallel version uses a carefully optimized protocol to achieve high transaction rates [371]. Software-parallel `silo` scales to 8.8× at 64 threads, 6.4× slower than Swarm. The reason is fine-grain 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-grain parallelism, which is mainly determined by the number of TPC-C warehouses. To quantify this effect, Figure 3-14 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-grain 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.

3.5.3 Swarm Analysis

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

Cycle breakdowns: Figure 3-15 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

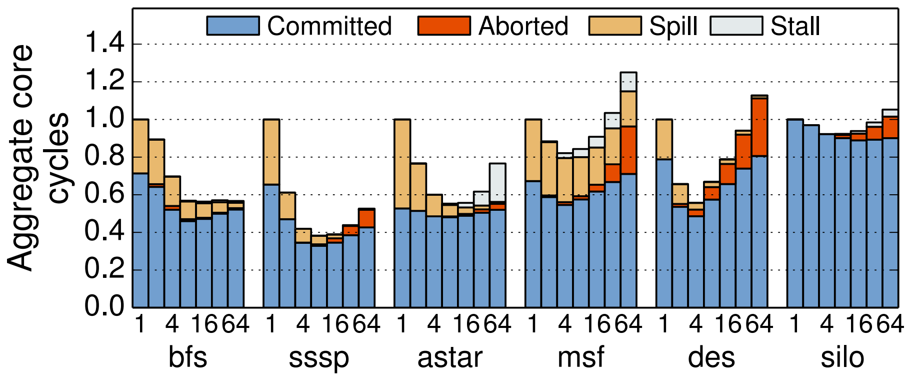


Figure 3-15: Breakdown of total core cycles for Swarm systems with 1 to 64 cores. Most time is spent executing tasks that are ultimately committed.

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).

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 3.3.7), 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 3-15 also shows the factors that contribute to super-linear scaling in Figure 3-12. 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

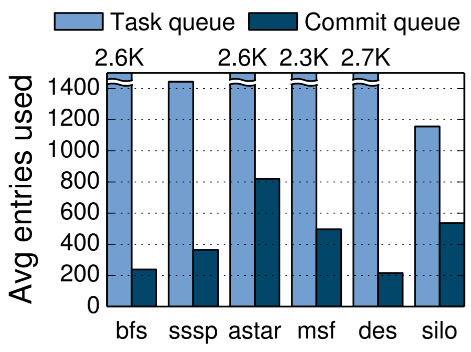


Figure 3-16: Average task and commit queue occupancies for 64-core Swarm.

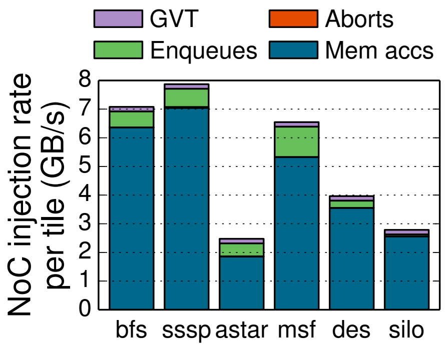


Figure 3-17: Breakdown of NoC traffic per tile for 64-core, 16-tile Swarm.

factor 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 3-16 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 3-17 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 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%.

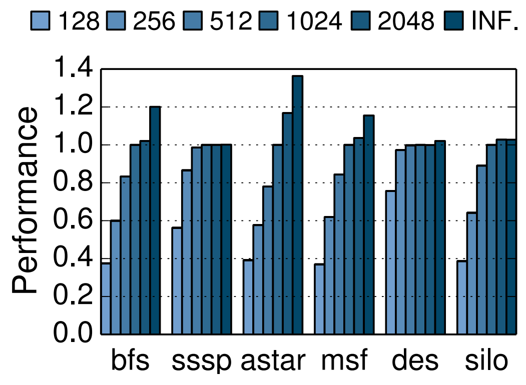
3.5.4 Sensitivity Studies

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

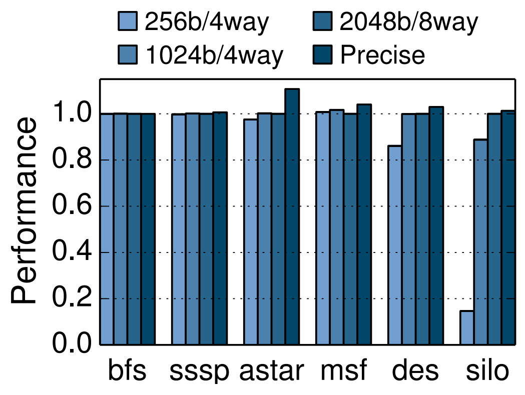
Commit queue size: Figure 3-18a shows the speedups of different applications as we sweep aggregate commit queue entries from 128 (8 tasks per tile) to unbounded; the default is 1024 entries. Commit queues are fundamental to performance: fewer than 512 entries degrade performance considerably. More than 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.

Bloom filter configuration: Figure 3-18b 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`.



(a) Commit queue size



(b) Bloom filter size

Figure 3-18: Sensitivity of 64-core Swarm to commit queue and Bloom filter sizes.

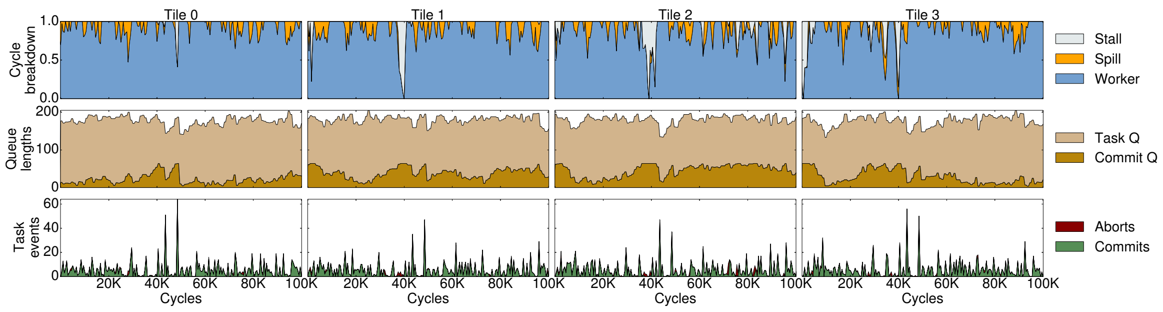


Figure 3-19: Execution trace of `astar` on 16-core (4-tile) Swarm over a 100 Kcycle interval: breakdown of core cycles (top), queue lengths (middle), and task commits and aborts (bottom) for each tile.

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 3-19 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 3-19 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 3-18a.

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 Additional Related Work

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 [24, 63, 124, 130, 149, 225, 273, 292, 388]. 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 [24, 225, 292]). 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 irregular applications.

Several TLS schemes expose timestamps to software for different purposes, such as letting the compiler schedule loop iterations in Stampede [348], letting the programmer relax sequential ordering [167] or speculate across barriers [169] in TCC, and supporting out-of-order spawn of speculative function calls in Renau et al. [309]. 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.

Prior work in fine-grain parallelism has developed a range of techniques to reduce task management overheads. Active messages lower the cost of sending tasks among cores [98, 276, 384]. Hardware task schedulers such as Carbon [221] lower overheads further for specific problem domains. GPUs [394] and Anton 2 [161] feature custom schedulers for non-speculative tasks. By contrast, Swarm implements speculative hardware task management for a different problem domain, ordered parallelism.

Prior work has developed shared-memory priority queues that scale with the number of cores [14, 231, 270, 311, 393], but they do so by relaxing priority order. This restricts them to benchmarks that admit order violations. For example, the correct discrete event simulation of a circuit requires hard ordering of tasks based on simulated time. Moreover, loose ordering can lose the work efficiency [49] of a parallel implementation: in `sssp`, threads often execute useless tasks far from the critical path [176, 177]. Nikas et al. [275] 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 priority queues, load-balanced through random enqueues, and uses speculation to maintain order.

Our execution model has similarities to parallel discrete-event simulation (PDES) [134]. PDES events run at a specific virtual time and can create other events, but cannot access arbitrary data, making them less general than Swarm tasks. Moreover, state-of-the-art PDES engines have overheads of tens of thousands of cycles per event [35], making them impractical for fine-grain tasks. Fujimoto proposed the Virtual Time Machine (VTM), tailored to the needs of PDES [143], which could reduce these overheads. However, VTM relied on an impractical memory system that could be indexed by address and time.

3.7 Summary

This chapter has presented Swarm, an architecture that unlocks abundant but hard-to-exploit ordered irregular 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. Programs leverage Swarm's execution model to convey new work to hardware as soon as it is discovered rather than in the order it needs to run, exposing a large amount of parallelism. As a result, Swarm achieves order-of-magnitude speedups on ordered irregular programs, which are key in emerging domains such as graph analytics, data mining, and in-memory databases. Swarm hardware could also support thread-level speculation and transactional execution with minimal changes.

Swarm's design challenges conventional wisdom in two ways. First, conventional wisdom says that task order constraints limit parallelism. However, we have shown that it is possible to maintain a large speculation window efficiently to extract ordered parallelism, wherein only true data dependences limit parallelism. Second, conventional wisdom says that speculation is wasteful, and designers should instead build non-speculative parallel systems. However, we have shown that, for a broad class of applications, speculation extracts abundant parallelism for moderate costs. Designers can trade this additional parallelism for efficiency in many ways (for example, through simpler cores or slower clock frequencies), more than offsetting the costs of speculation. In other words, speculation can yield a net efficiency gain and enable more applications to exploit the performance potential of multi-billion-transistor chips.

CHAPTER 4

Spatial Hints: Data-Centric Execution of Speculative Parallel Programs

This work was conducted in collaboration with Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. The ideas for spatial-hint task mapping and serialization were developed collaboratively. This thesis contributes the spatial-hint selection patterns across applications, and the implementation and study of fine-grain tasks. This thesis also contributes to the development of applications, and the architectural simulator.

Speculative parallelization, e.g., through Swarm, thread-level speculation (TLS), or hardware transactional memory (HTM), has two major benefits over non-speculative parallelism: it uncovers abundant parallelism in many challenging applications [169] (Chapter 3) and simplifies parallel programming [284, 316]. However, even with scalable versioning and conflict detection techniques, speculative systems scale poorly beyond a few tens of cores. A key reason is that these systems *do not exploit much of the locality available among speculative tasks*.

To scale, 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 abundant prior work has developed programming models to convey locality [8, 38,73,353,398], and runtimes and schedulers to exploit it [6,51,77,188,269,340,400].

However, most prior work in speculative parallelization has ignored the need for locality-aware parallelism. In TLS, speculative tasks are executed by available cores without regard for locality [168,308,344]; Swarm sends new tasks to randomly chosen tiles for load balance; and conventional HTM programs are structured as threads that execute a fixed sequence of transactions. Prior work has observed that it is beneficial to structure transactional code into tasks instead, and has proposed transactional task schedulers that *limit concurrency* to reduce aborts under high contention [19,21,44,45, 110,116,199,320,402]. Limiting concurrency suffices for small systems, but scaling to hundreds of cores also requires solving the *spatial mapping* problem: speculative tasks must be mapped across the system to minimize data movement.

To our knowledge, no prior work has studied the spatial mapping problem for speculative architectures. This may be because, at first glance, spatial mapping and speculation seem to be at odds: achieving a good spatial mapping requires knowing the data accessed by each task, but the conventional advantage of speculation is precisely that the programmer or compiler need not or does 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 just before 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 reducing the misspeculation rate and making most remaining conflicts local.

To exploit this insight, we present *spatial hints*, a technique that uses program knowledge to achieve high-quality task mappings (Section 4.2). A hint is an abstract integer, given at run time when a task is created, that denotes the data that the task is likely to access. We show it is easy to modify programs to convey locality through hints. We enhance Swarm to exploit hints by sending tasks with the same hint to the same tile and running them serially.

We then analyze how task structure affects the effectiveness of hints (Section 4.4). We find that fine-grain tasks access less data, and more of that data is known at task creation time, making hints more effective. Although programs with fine-grain tasks perform more work and put more pressure on scheduling structures, hints make fine-grain tasks a good tradeoff by reducing memory stalls and conflicts further. We show that certain programs can be easily restructured to use finer-grain tasks (Section 4.4), improving performance by up to 2.7×.

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 (Section 4.5). Unlike non-speculative load balancers, the signals to detect imbalance are different with speculation (e.g., tiles do not run out of tasks, but run tasks that are likely to abort), requiring a different approach. Our load balancer improves performance by up to a further 27%.

In summary, this chapter presents four novel contributions:

- Spatial hints, a technique that conveys program knowledge to achieve high-quality spatial task mappings.
- Simple hardware mechanisms to exploit hints by sending tasks likely to access

the same data to the same place and running them serially.

- An analysis of the relationship between task granularity and locality, showing that programs can often be restructured to make hints more effective.
- A novel data-centric load-balancer that leverages hints to redistribute tasks without hurting locality.

Together, these techniques make speculative parallelism practical on large-scale systems: at 256 cores, hints achieve near-linear scalability on nine challenging applications, outperform the baseline Swarm random scheduler by 3.3× gmean and by up to 16×, and outperform a work-stealing scheduler by a wider margin. Hints also make speculation far more efficient, reducing wasted work by 6.4× and network traffic by 3.5× on average.

4.1 Motivation

We demonstrate the benefits of spatial task mapping on Swarm (Chapter 3). Swarm is a strong baseline for two key reasons. First, Swarm's task-based, timestamp-ordered execution model is general: it finds parallelism among ordered and unordered tasks, subsuming both TLS and TM, and allows more ordered programs to be expressed than TLS. This allows us to test our techniques with a broader range of speculative programs than alternative baselines. Second, Swarm focuses on efficiently supporting fine-grain tasks, and includes hardware support for task creation and queuing. This allows us to study the interplay between task granularity and spatial hints more effectively than alternative baselines with software schedulers, which are limited to coarse-grain tasks.

We use the discrete-event simulation (des) of a digital circuit under different task schedulers as an example to motivate the need for spatial task mapping. Listing 4.1

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

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

Listing 4.1: Swarm implementation of discrete event simulation for digital circuits.

shows the Swarm implementation of `des`. It defines one task function, `desTask`, which simulates a signal toggling at a gate input at a particular simulated time (e.g., in picoseconds). If that input toggle causes the gate output to toggle, `desTask` enqueues child tasks for all the gates connected to this output. The program initiates the simulation by enqueuing a task for each input waveform, then calling `swarm::run`. We simulate Swarm systems of up to 256 cores, as shown in Figure 4-2 (see Section 4.3.1 for methodology details). We compare four schedulers:

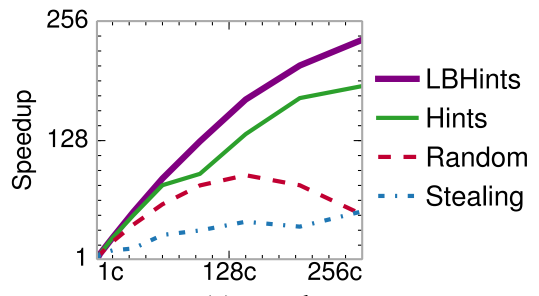
- Random is Swarm's default task mapping strategy. New tasks are sent to a random tile for load balance.
- Stealing is an *idealized work-stealing* scheduler, the most common scheduler in non-speculative programs [6, 51]. New tasks are enqueued to the local tile, and tiles that run out of tasks steal 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 4.6.2). 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 logically maps each gate in the simulated circuit to a specific, statically chosen tile. New tasks that operate on a particular gate are sent to its logically mapped tile (Section 4.2). LBHints enhances Hints by periodically remapping gates across tiles to equalize their load (Section 4.5).

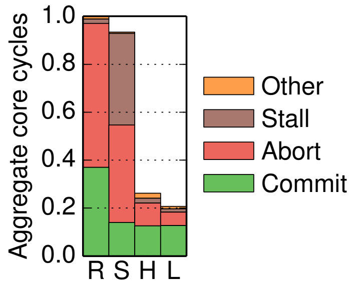
Two factors make spatial mapping in `des` possible. First, each task operates on a single gate. Second, this gate is known at run time when the task is created. As we will see later, good spatial mappings are possible even when these conditions are not completely met (i.e., tasks access multiple pieces 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 4-1a 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 4-1b yields further insights into these differences. The height of each bar in Figure 4-1b 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



(a) Speedup



(b) Cycle breakdown

Figure 4-1: 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.

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.

4.2 Spatial Task Mapping with Hints

We now present *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. We present the API and ISA extensions to support hints, describe the microarchitectural mechanisms to exploit hints, and show how to apply hints to a variety of benchmarks.

4.2.1 Hint API and ISA Extensions

We extend the `swarm::enqueue` function (Section 3.3.1) with one field for the spatial hint:

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

This 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.

As with Swarm code enqueues tasks with an `enqueue_task` instruction that takes the function pointer, timestamp, and arguments in registers. We employ unused bits in the 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.

4.2.2 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 a request to the selected tile to enqueue the task descriptor (Section 3.3.2). `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.

4.2.3 Adding Hints to Benchmarks

We add hints to a diverse set of nine benchmarks. Table 4.1 summarizes their provenance, input sets, and the strategies used to assign hints to each benchmark.

Seven of our benchmarks are ordered, including `bfs`, `sssp`, `astar`, `des`, and `silo` from Chapter 3, and this chapter adds two others:

- **`color`** uses the largest-degree-first heuristic [391] to assign distinct colors to adjacent graph vertices. This heuristic produces high-quality results and is thus frequently used, but it is hard to parallelize.
- **`nocsim`** is a detailed network-on-chip simulator derived from GARNET [7]. Each task simulates an event at a component of a router.

We port `color` and `nocsim` to Swarm from existing serial implementations. As in Chapter 3, these applications do not change the amount of work relative to the serial code. As shown in Table 4.1, at 1 core, Swarm implementations outperform tuned serial versions in all cases except `bfs`, where 1-core Swarm is 18% slower.

We also port two unordered benchmarks from STAMP [257]:

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

We implement transactions as tasks of equal timestamp, so that they can commit in any order. As in prior work in transaction scheduling [19, 402] (Section 4.6), we break the original threaded code into tasks that can be scheduled asynchronously and generate children tasks as they find more work to do.

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: each task operates on one vertex and visits its neighbors in some programmer-specified order. For example, Listing 4.2 shows the single task function of `sssp` from Listing 3.3. 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

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

Listing 4.2: A hint-tagged Swarm task for Dijkstra's `sssp` algorithm.

distance to the source.

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 `v->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-grain versions make hints much more effective.

Object IDs: In `des` and `nocsim` each task operates on one system component: a logic gate (Listing 4.1), 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

	Source	Input	Swarm 1-core		Task	Hint patterns
			Run-time	Perf vs serial	Funcs	
bfs	PBFS [230]	hugetric-00020 [26, 100]	3.59 Bcycles	−18%	1	Cache line of vertex
sssp	Galois [289]	East USA roads [1]	3.21 Bcycles	+33%	1	Cache line of vertex
astar	(Chapter 3)	Germany roads [280]	1.97 Bcycles	+1%	1	Cache line of vertex
color	[174]	com-youtube [233]	1.65 Bcycles	+54%	3	Cache line of vertex
des	Galois [289]	csaArray32	1.92 Bcycles	+70%	8	Logic gate ID
nocsim	GARNET [7]	16x16 mesh, tornado traffic	22.37 Bcycles	+68%	10	Router ID
silo	[371]	TPC-C, 4 whs, 32 Ktxns	2.83 Bcycles	+16%	16	(Table ID, primary key)
genome	STAMP [257]	-g4096 -s48 -n1048576	2.30 Bcycles	+1%	10	Elem addr, map key, NO/SAMEHINT
kmeans	STAMP [257]	-m40 -n40 -i rnd-n16K-d24-c16	8.56 Bcycles	+2%	5	Cache line of point, cluster ID

Table 4.1: Benchmark information: source implementations, inputs, run-times on a 1-core Swarm system, 1-core speedups over tuned serial implementations, number of task functions, and hint patterns used.

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 4.1.

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 we 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`).

4.3 Evaluation of Spatial Hints

4.3.1 Experimental Methodology

Modeled system: We use a cycle-accurate, event-driven simulator based on Pin [242, 282] to model Swarm systems of up to 256 cores, as shown in Figure 4-2, with parameters in Table 4.2. 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 [101], 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 also simulate smaller systems with square meshes ($K \times K$ tiles for $K \leq 8$). 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.

Whereas Chapter 3 models simple IPC-1 cores, in this chapter and all subsequent

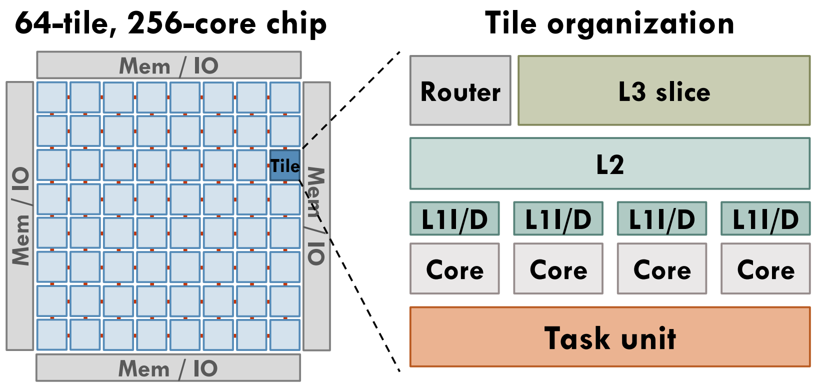


Figure 4-2: Swarm 256-core chip and tile configuration.

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 in-order scoreboarded (stall-on-use), functional unit latencies as in Nehalem [323], 4-entry load and store 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 [211] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	8×8 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [392])
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 [66] 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 4.2: Configuration of the 256-core system.

chapters, 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 [323]. Cores are scoreboarded and stall-on-use, permitting multiple memory requests in flight.

Benchmark configuration: Table 4.1 reports the input sets used. We compile benchmarks with `gcc 6.1`. All benchmarks have 1-core run-times of over 1.6 billion cycles (Table 4.1). Benchmarks from Chapter 3 use the same inputs. `color` operates on a YouTube social graph [233]. `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. Like in Section 3.4, we report results for the full parallel region.

4.3.2 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-*

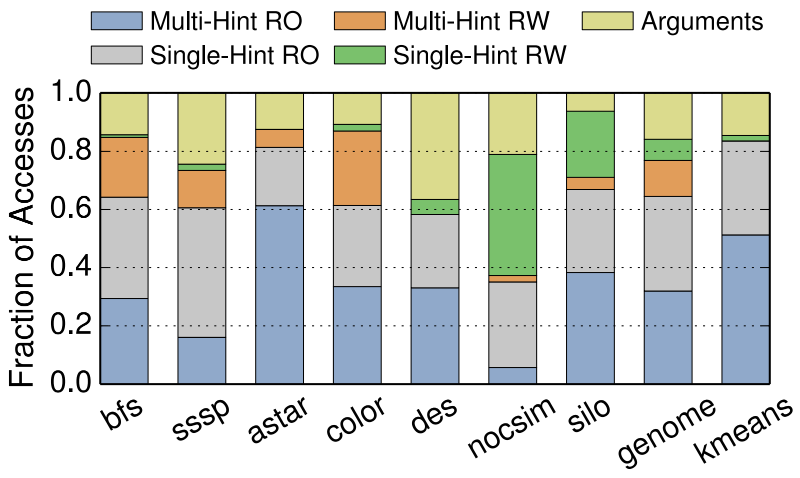


Figure 4-3: Classification of memory accesses.

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 4-3 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).

Figure 4-3 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-grain versions in Section 4.4 will improve this to 8 out of 9). We now evaluate the impact of these results on performance.

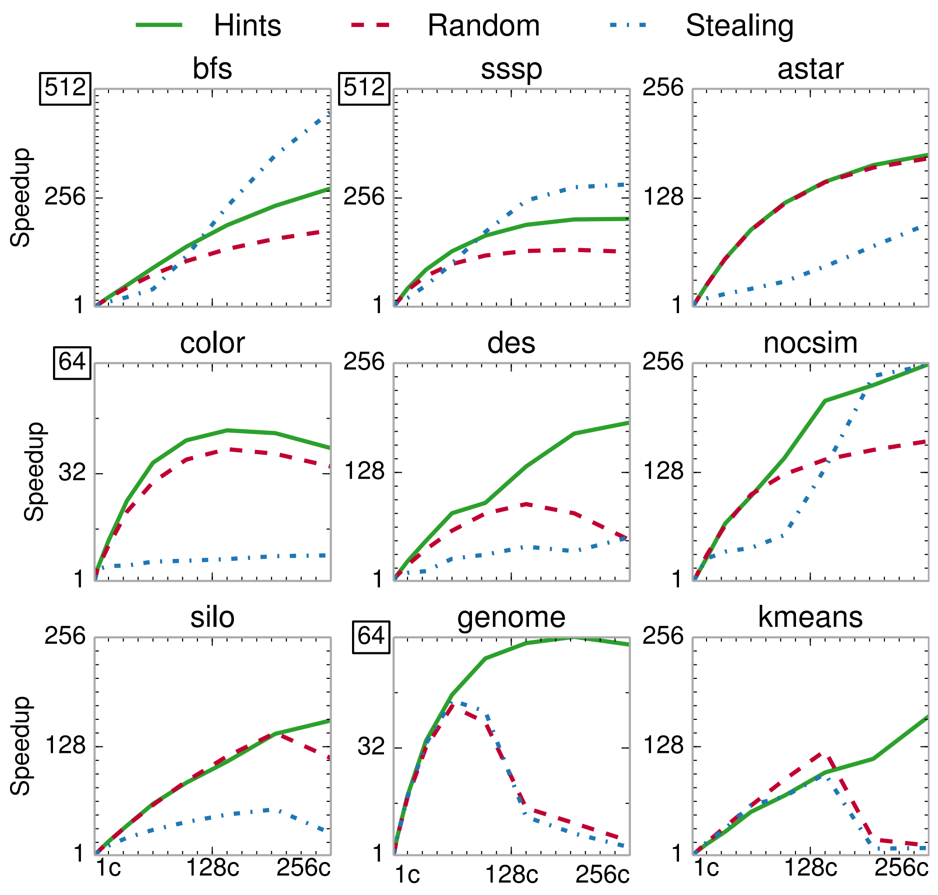


Figure 4-4: Speedup of different schedulers from 1 to 256 cores, relative to a 1-core system. We simulate systems with $K \times K$ tiles for $K \leq 8$.

4.3.3 Comparison of Schedulers

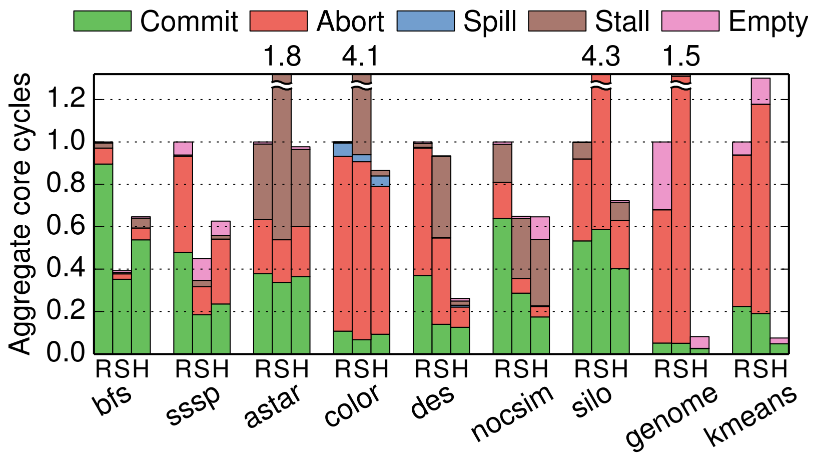
Figure 4-4 compares the scalability of the Random, Stealing, and Hints schedulers on 1–256-core systems, similar to Figure 4-1a. As in Section 3.5, we keep *per-core* queue and L2/L3 capacities constant, capturing performance per unit area.[2]

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.

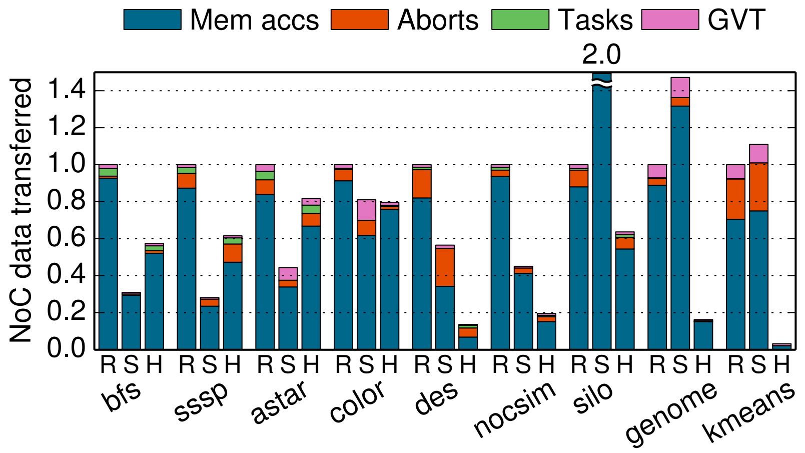
Figure 4-5 gives more insight into these results by showing core cycle and network

[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.

[2]Section 3.5.3 details the contributions of scaling queue/cache capacities; the same trends hold.



(a) Breakdown of total core cycles



(b) Breakdown of total NoC data transferred

Figure 4-5: 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.

traffic breakdowns at 256 cores. Each bar of Figure 4-5a 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 4-5b 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 4.3.2).

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 4-4, and *(ii)* empty stalls are the remaining overhead at 256 cores. Hint-based load balancing addresses this problem (Section 4.5). 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 4-5a.

Stealing: Stealing shows disparate performance across benchmarks, despite careful tuning and idealizations (Section 4.1). 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 vertices in the same tile. Because each task accesses a vertex and its neighbors (Listing 4.2), 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.

4.4 Improving Locality and Parallelism with Fine-Grain Tasks

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

For example, consider the coarse-grained implementation of sssp in Listing 4.2. Each task may read and write the vertex distances of several neighbors; conversely each distance is read and written by multiple tasks. This renders hints ineffective for read-write data. Instead, Listing 4.3 shows an alternative version of sssp where each task operates on the data (distance and neighbor list) of a *single* vertex.

Instead of setting the distances of all its neighbors, this task creates one child task per neighbor. Each task accesses its own distance. This transformation generates substantially more tasks, as each vertex is visited once for every incident edge. In a serial or parallel version with software scheduling, the coarse-grain 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

```
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 4.3: Fine-grain sssp implementation.

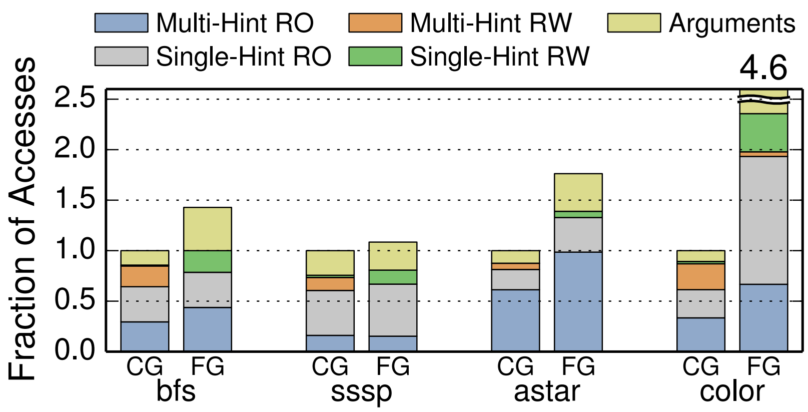


Figure 4-6: Classification of memory accesses for coarse-grain (CG) and fine-grain (FG) versions.

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 (*pulls* [80, 270]) data from all neighboring vertices, then updates its own. Our fine-grain version reverses this operation to read and update local vertex data in one type of task, then send other tasks to *push* [80, 270] updates to each neighbor; every task reads or writes mutable data from 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.

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

Effectiveness of Hints: Figure 4-6 compares the memory accesses of coarse-grain (CG) and fine-grain (FG) versions. Figure 4-6 is similar to Figure 4-3, but bars are normalized to the CG version, so the height of each FG bar denotes how many more accesses it makes. Figure 4-6 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).

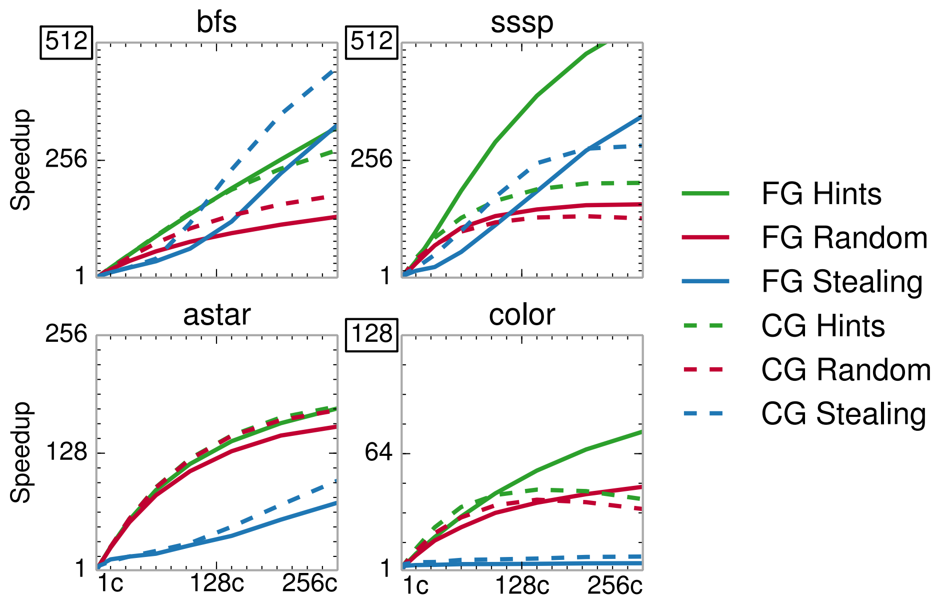
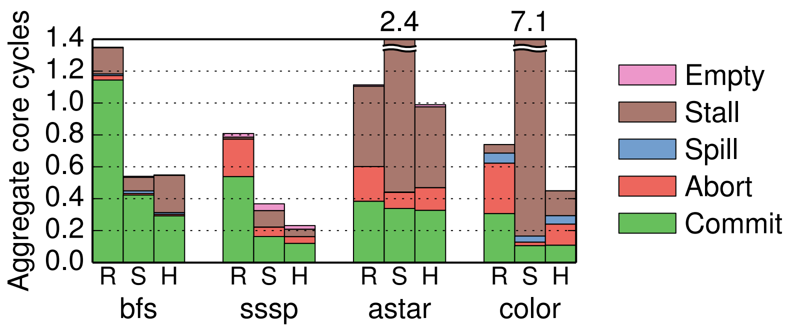
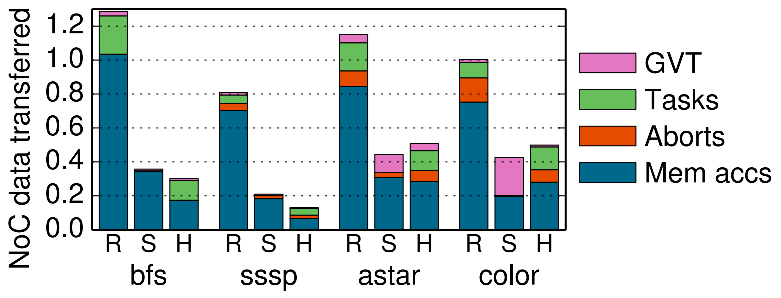


Figure 4-7: Speedup of fine-grain (FG) and coarse-grain (CG) versions, relative to CG versions on a 1-core system.



(a) Breakdown of total core cycles



(b) Breakdown of total NoC data transferred

Figure 4-8: Breakdown of (a) core cycles and (b) NoC data transferred in fine-grain versions at 256 cores, under Random, Stealing, and Hints. Each bar is normalized to the coarse-grain version under Random (as in Figure 4-5).

4.4.1 Evaluation

Figure 4-7 compares the scalability of CG and FG versions under the three schedulers. Speedups are relative to the CG versions at one core. Figure 4-8 shows cycle and network traffic breakdowns, with results normalized to CG under Random (as in Figure 4-5). 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 4-8a shows that Hints improve locality (fewer committed cycles) and reduce aborts. As a result, FG versions under Hints incur much lower traffic (Figure 4-8b), 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 4-8a). Nonetheless, FG improves efficiency and reduces traffic by 61% over CG under Hints (Figure 4-8b).

`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 4-8a), outperforming CG under Hints by 93%.

Finally, comparing the relative contributions of tasks sent in Figure 4-8b vs. Figure 4-5b 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.

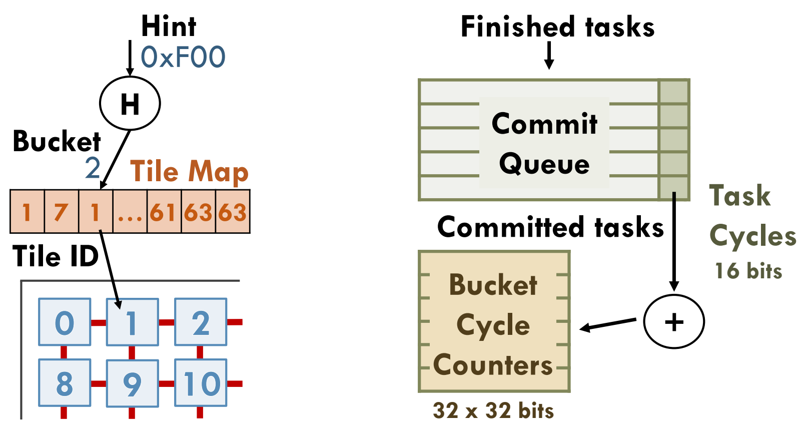
In summary, fine-grain versions substantially improve the performance and efficiency of Hints. This is not always the case for Random or Stealing, as they do not exploit the locality benefits of fine-grain tasks.

4.5 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.

Instead, we have found that the number of committed cycles is a better proxy for



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

Figure 4-9: Hardware modifications of hint-based load balancer.

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 4-9(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 4-9(b). First, like the hashed hint (Section 4.2.2), 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).

4.5.1 Evaluation

Figure 4-10 reports the scalability of applications with our hint-based load balancer, denoted LBHints. LBHints improves performance on four applications, and neither

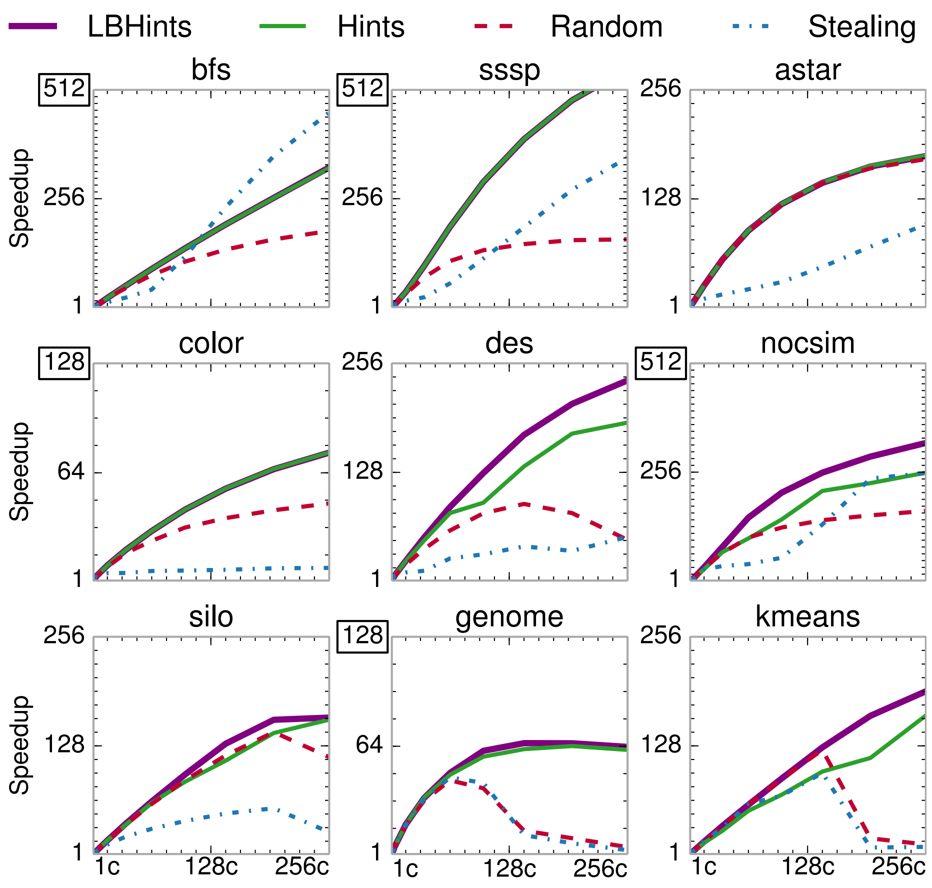


Figure 4-10: Speedup of Random, Stealing, Hints, and LBHints schedulers from 1 to 256 cores, relative to a 1-core system. For applications with coarse- and fine-grain versions, we report the best-performing version for each scheme.

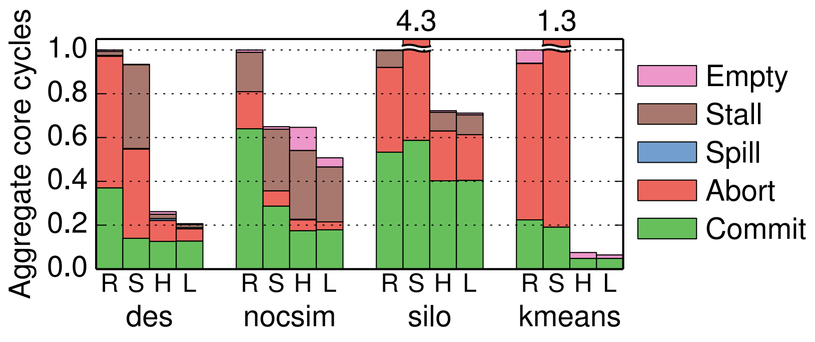


Figure 4-11: Breakdown of total core cycles at 256 cores under Random, Stealing, Hints, and LBHints.

helps nor hurts performance on the other five.

In des, LBHints outperforms Hints by 27%, scaling to 236×. As described in Section 4.1, in des load imbalance causes aborts as some tiles run too far ahead; LBHints's gains come from reducing aborts, as shown in Figure 4-11. 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.

4.5.2 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-grain versions from Section 4.4 instead of their coarse-grain counterparts, Hints scales to 179× (while Random scales to 62× only); and LBHints scales to 193× gmean.[3]

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

4.6 Additional Related Work

4.6.1 Scheduling in Speculative Systems

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

Ordered parallelism: TLS schemes dispatch tasks to threads as they become available, without concern for locality [150, 168, 308, 309, 344, 349]. TLS schemes targeted systems with few cores, but cache misses hinder TLS scalability even at small scales [145].

Unordered parallelism: TM programs are commonly structured as threads that execute a fixed sequence of transactions [69, 169, 262]. 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 [19, 21, 44, 45,110,116,320,402]. Prior transactional schedulers focus on *limiting concurrency*, not spatial task mapping. These schemes are either reactive or predictive. ATS [402], CAR-STM [114], and Steal-on-Abort [19] serialize aborted transactions after the transaction they conflicted with, avoiding repeated conflicts. PTS [44], BFGTS [45], Shrink [116], and HARP [21] 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. [219] 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 [94] 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 [57, 183, 318]. Some of these schemes, like ClusterSTM [57], 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 [57], so load balancing is not an issue.

4.6.2 Scheduling in Non-Speculative Systems

In contrast to speculative models, prior work for non-speculative parallelism has developed many techniques to improve locality, often tailored to specific program traits [6, 8, 38, 51, 73, 77, 188, 269, 340, 398, 400]. Work-stealing [6, 51] 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 4.3). Due to its low overheads, work-stealing is the foundation of most parallel runtimes [121, 196, 217], which extend it to improve locality by stealing from nearby cores or limiting the footprint of tasks [6, 164, 324, 400], or to implement priorities [231, 271, 324]. Prior work within the Galois project [176, 231, 289] 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 [271, 293]. 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 [221, 326] and domain-specific schedulers [161, 394].

4.7 Summary

This chapter has presented spatial hints, a general technique that leverages application-level knowledge to achieve high-quality spatial task mappings in speculative programs. A hint is an abstract value, given at task creation time, that denotes the data likely to be accessed by a task. We have enhanced Swarm from Chapter 3 to exploit hints by *(i)* running tasks likely to access the same data on the same tile, *(ii)* serializing tasks likely to access the same data, and *(iii)* balancing work across tiles in a locality-aware fashion. We have also studied the relationship between task granularity and locality, and shown that programs can often be restructured to use finer-grain tasks to make hints more effective.

Together, these techniques make speculative parallelism practical on large-scale systems: at 256 cores, the baseline Swarm system accelerates nine challenging applications by 5–180× (gmean 58×). With the techniques in this chapter, speedups increase to 64–561× (gmean 193×). Beyond improving gmean performance by 3.3×, these techniques make speculation more efficient, reducing aborted cycles by 6.4× and network traffic by 3.5× on average.

CHAPTER 5

Espresso and Capsules: Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism

This work was conducted in collaboration with Victor A. Ying, Suvinay Subramanian, Hyun Ryong Lee, Joel Emer, and Daniel Sanchez. The Espresso execution model, exception model, and Capsules semantics were developed collaboratively. This thesis contributes the safe dispatch of MAYSPEC and NONSPEC tasks, their conflict detection concerns, and the promotion mechanism. This thesis also contributes to the development of applications, and the architectural simulator.

Chapter 3 and Chapter 4 showed that speculative execution is an important tool to extract abundant fine-grain irregular parallelism. However, even applications that need speculation to scale have some work that is best executed non-speculatively. For example, some tasks are straightforward to synchronize with locks or atomic instructions, and running them speculatively adds overhead and needless aborts. Moreover, non-speculative parallelism is required to perform irrevocable actions, such as disk or network I/O, in parallel.

Ideally, systems should support composition and coordination of speculative and non-speculative tasks, and allow those tasks to safely synchronize access to the same data. Unfortunately, prior techniques fall short of this goal. All prior hardware techniques to combine speculative and non-speculative parallelism have been done in the context of hardware transactional memory (HTM) systems. HTM supports both speculative (transactional) and non-speculative (non-transactional) code. But HTM *lacks shared synchronization mechanisms*, so speculative and non-speculative code cannot easily access shared data [119, 382]. Moreover, most HTMs provide *unordered* execution semantics that miss many opportunities for parallelization. Systems that use speculation to unlock *ordered* parallelism, such as thread-level speculation (TLS), Swarm, and HTMs with programmer-controllable commit order [69, 169, 291], *disallow non-speculative parallelism*. In these systems, all tasks except the earliest active one execute speculatively.

The goal of this chapter is to bring the benefits of non-speculative execution to systems that support ordered irregular parallelism. This is not merely a matter of adapting HTM techniques. Unordered and ordered speculation systems address different needs and need different mechanisms (Section 5.1). To meet this goal, we contribute two main techniques.

The first contribution is *Espresso*, an expressive execution model that generalizes Swarm and spatial hints for speculative and non-speculative parallelism (Section 5.2). In Espresso, all work happens within tasks, which can run speculatively or non-speculatively. Tasks can create children tasks that run in either mode. Like Swarm, Espresso efficiently supports fine-grain tasks of a few tens of instructions each, so many tasks access a single piece of data, which is known when the task is created. To exploit this, Espresso elevates Swarm timestamps and spatial hints into synchronization mechanisms that efficiently coordinate speculative and non-speculative tasks. Moreover, Espresso lets the system decide whether to run certain tasks speculatively or non-speculatively, reaping the efficiency of non-speculative parallelism when it is plentiful, while exploiting speculative parallelism when needed to scale.

The second contribution is *Capsules*, a technique that lets speculative tasks selectively bypass hardware-managed speculation (Section 5.3) to rely on software-managed techniques that avoid serialization or conflicts. Among other uses, this enables scalable system services and concurrent system calls. Prior work in HTM has proposed escape actions [64, 264, 411] and open-nested transactions [251, 264, 266] to achieve similar goals. Unfortunately, these mechanisms are incompatible with many modern speculative systems. Specifically, they are incompatible with *speculative forwarding* which allows speculative tasks to read data written by uncommitted tasks (Section 2.4.2, Section 3.3.4). Forwarding is critical for extracting ordered parallelism, but causes tasks to lose data and control-flow integrity (Section 5.1.3). Capsules solve this problem by implementing a safe mechanism to transition out of hardware-managed speculation and by protecting certain memory regions from speculative accesses. Unlike prior techniques, Capsules can be applied to any system for speculative system, even if speculative tasks can lose data or control-flow integrity.

These contributions improve performance and efficiency, and enable new capabilities. We implement Espresso and Capsules atop Swarm (Section 5.4) and evaluate them on a diverse set of challenging applications (Section 5.5). At 256 cores, Espresso outperforms non-speculative-only execution by gmean 6.9× and Swarm's speculative-only execution by gmean 22%. Capsules enable the efficient implementation of important system services, like a scalable memory allocator that improves performance by up to 69×, and allow speculative tasks to issue concurrent system calls, e.g., to fetch data from disk.

5.1 Motivation

We present three case studies that show the need to combine speculative and non-speculative parallelism. Espresso subsumes prior speculative execution models (HTM, TLS, and Swarm), so these case studies use our Espresso implementation (Section 5.4), which does not penalize programs that do not use its features.

5.1.1 Speculation Benefits Are Input-Dependent

Just as Dijkstra's algorithm for single-source shortest paths (`sssp`) motivated the need for speculative parallelism in Chapter 3, it also aptly illustrates the *tradeoffs* between speculative and non-speculative parallelism. `sssp` admits a non-speculative, one-distance-at-a-time parallelization [107, 230, 253]. At any given time, the system only processes tasks with the lowest unprocessed distance; these create tasks with higher distances. After all tasks for the current distance have finished, cores wait at a barrier and collectively move on to the next unprocessed distance. Since multiple same-distance tasks may visit the same vertex, each task must use proper synchronization to ensure safety.

This non-speculative `sssp` works well if the graph is shallow and there are many vertices with the same distance to the source. However, weighted graphs often have very few vertices per distance, so non-speculative `sssp` will find little work between barriers. In this case, scaling `sssp` requires exploiting ordered parallelism, processing tasks across multiple distances simultaneously. While most tasks are independent, running dependent (same-vertex) tasks out of order will produce incorrect results. Hence, exploiting ordered parallelism requires speculative execution, running tasks out of order and committing them in order. The Swarm architecture can do this, but it runs all tasks (except the earliest active one) speculatively (Chapter 3).

Neither strategy is always the best. Figure 5-1 compares the speedups of the non-speculative (Non-spec) and fully speculative (All-spec) versions of `sssp` on two graphs: `usa`, a graph of Eastern U.S. roads, and `cage`, a graph arising from DNA electrophoresis. Both versions leverage Espresso's hardware-accelerated task scheduling and locality-aware execution (see Section 5.5.1 for methodology). At 256 cores, on `usa` All-spec is 248× faster than Non-spec, which barely scales because there are few vertices per distance. By contrast, `cage` is a shallow unit-weight graph with about 36000 vertices per distance, so Non-spec outperforms All-spec by 21%, as it does not incur the overheads

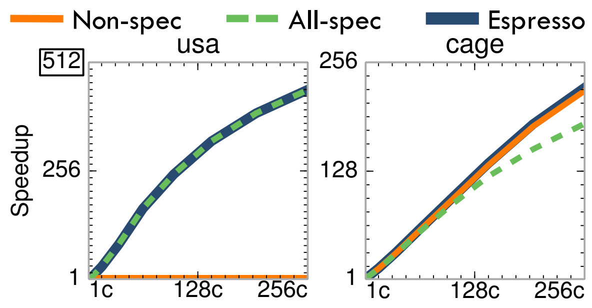


Figure 5-1: Speedup of three versions of sssp on 1–256 cores for two graphs. Total cache and queue capacities grow with core count (Section 5.5.1), causing superlinear speedup on usa.

of speculative execution.

These results show that forcing the programmer to choose whether to use all-or-nothing speculation is undesirable. The best way to parallelize sssp is a hybrid strategy: all lowest-distance tasks should run non-speculatively, relying on cheaper synchronization mechanisms to provide mutual exclusion, while higher-distance tasks should run speculatively to exploit ordered parallelism. But this requires *the same task* to be runnable in either mode, which is not possible in current systems.

Espresso provides the mechanisms needed for this hybrid strategy (Section 5.2). First, it provides two synchronization mechanisms, timestamps and locales, that have consistent semantics across speculative and non-speculative tasks. Second, it lets the system choose whether to speculate or not, based on the amount of available parallelism. Figure 5-1 shows that the Espresso version of sssp achieves the best performance on both graphs, because it only uses speculative parallelism when non-speculative parallelism is insufficient.

5.1.2 Combining Speculative and Non-Speculative Tasks

Even in applications that need speculative parallelization, some tasks are best run non-speculatively. Consider des, our discrete event simulator for digital circuits (Listing 4.1). Each des task evaluates the effects of toggling a gate input at a particular simulated time; if the gate's output toggles, the task creates new tasks for gates connected to this output. As with sssp, ordered speculation enables des to scale to hundreds of cores (Section 4.3).

However, a common feature in logic simulators is to log the waveforms of intermediate signals. We extend des so that each simulation task that causes a toggle creates a separate logging task that writes the event to a per-core in-memory log.

Although simulation tasks require ordered speculation to scale, there is no good reason for logging tasks to speculate. Logging is trivial to synchronize. Prior architectures for ordered parallelism, however, run all tasks speculatively. This causes abort cascades: if a simulation task aborts, its child logging task aborts, and this in turn causes all logging tasks that have later written to the same log to abort.

The right strategy is to let each (ordered) speculative simulation task launch an (unordered) non-speculative logging task. These logging tasks can run in parallel, but a logging task runs only after its speculative parent commits, avoiding mispeculation. If its parent aborts, it is discarded. Espresso enables this approach.

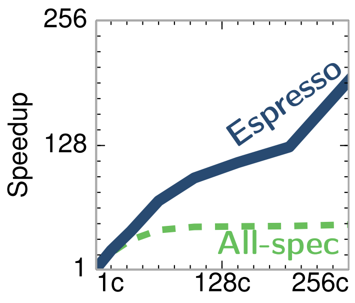


Figure 5-2: `des` speedup on 1–256 cores, with speculative and non-speculative logging tasks.

Figure 5-2 compares the speedups of using speculative and non-speculative logging in `des`. Speculative logging causes needless aborts that limit `des`'s performance beyond 64 cores. At 256 cores, non-speculative logging is 4.1× faster.

Prior work in HTM has proposed to let transactions register *commit handlers* that run only at transaction commit [251]. Espresso generalizes this idea to let any task create speculative or non-speculative children.[1] Additionally, Espresso's implementation requirements are different: unordered HTMs commit transactions immediately after running, while ordered tasks can stay speculative many cycles after they finish.

5.1.3 Software-Managed Speculation Improves Parallelism

When a speculative task produces output that is not immediately needed, it can create a non-speculative child task (Section 5.1.2). However, a speculative task often needs to use the results of some action, such as allocating memory, that is best done without hardware speculation.

Prior work in HTM has proposed *escape actions* for this purpose [64, 264, 411]. Escape actions let a transaction temporarily turn off hardware conflict detection and version management and run arbitrary code, including system calls. An escape action can register an abort handler that undoes its effects if the enclosing transaction aborts. For example, a transaction can use escape actions to allocate memory from a conventional thread-safe allocator, avoiding conflicts on allocator metadata. The escape action's abort handler frees the allocated memory.

Unfortunately, escape actions and similar mechanisms, such as open-nested transactions [251, 264, 266], are incompatible with architectures for ordered parallelism and many recent HTMs. These architectures perform *speculative data forwarding* [25, 137, 199, 291, 296, 302, 344, 349], which lets tasks access data written by earlier, uncommitted tasks.[2] Speculative forwarding is crucial because ordered tasks may take a long time to commit. Without speculative forwarding, many ordered algorithms scale

[1] For example, a non-speculative child with the same timestamp as its speculative parent is equivalent to a transactional commit handler.

[2] We follow TLS terminology [291, 344, 349]; in software TMs, lack of speculative forwarding is referred to as *opacity* [97, 163].

poorly [277, 349] (e.g., `des` is 5× slower at 256 cores).[3] However, letting tasks access uncommitted state means they may read inconsistent data, and lose data and control-flow integrity (e.g., by dereferencing or performing an indirect jump to an invalid pointer). This makes escape actions unsafe: a mispeculating task can begin a malformed escape action, or an escape action might read temporarily clobbered data. Such an escape action could perform actions that cannot be undone by an abort handler.

Without escape actions, prior ordered speculative systems are forced to run memory allocation routines speculatively, and suffer from spurious conflicts on allocator metadata. Figure 5-3 shows the performance of `des` when linked with TCMalloc [154], a state-of-the-art memory allocator that uses thread-local structures. `des` scales poorly with TCMalloc, achieving a speedup of 23× at 100 cores and declining significantly at higher core counts, where it is overwhelmed with aborts.

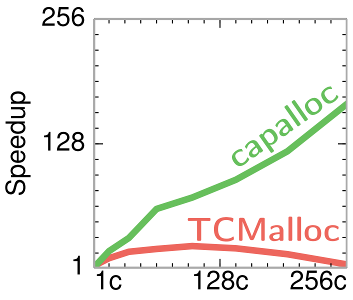


Figure 5-3: `des` speedup on 1–256 cores, with different allocators.

Capsules (Section 5.3) solve this problem by providing a general solution for providing protected access to software-managed speculation. The program prespecifies a set of *capsule functions*, and the system guarantees that speculative tasks can only disable hardware speculation by invoking these functions. This prevents mispeculating tasks from invoking arbitrary actions. Moreover, capsule functions have access to memory that is not conflict-checked and is protected from accesses by speculative tasks. Figure 5-3 shows the performance of `capalloc`, a memory allocator similar to TCMalloc that implements all allocation routines, such as `malloc`, in capsule functions and uses lock-based synchronization. `capalloc` makes `des` scale well, outperforming TCMalloc by 39× at 256 cores.

5.2 Espresso Execution Model

Espresso programs consist of tasks that run speculatively or non-speculatively. All tasks can access shared memory and make arbitrary system calls. Espresso provides two synchronization mechanisms: *timestamps* to convey order requirements and *locales* to convey mutual exclusion and locality information. Each task can optionally be given a timestamp and a locale. Timestamps and locales have common semantics among speculative and non-speculative tasks, allowing tasks running in either *mode* to coordinate accesses to shared data.

Espresso supports three task *types* that control speculation: `SPEC` tasks always run speculatively, `NONSPEC` tasks always run non-speculatively, and `MAYSPEC` tasks may run

[3] Without speculative forwarding, systems must stall [262] or abort [56, 401] tasks that access uncommitted data.

```
void ssspTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::enqueue<MAYSPEC>(&ssspTask,
          /*timestamp=*/ dist + weight(v,n),
          /*locale=*/ n->id, n);
  }
}

void main() {
  [...] /* Set up graph and initial values */
  espresso::enqueue<MAYSPEC>(&ssspTask,
     0, source->id, source);
  espresso::run();
}
```

Listing 5.1: Espresso implementation of Dijkstra's `sssp` algorithm.

in either mode. All tasks can create children tasks of any type.

We expose these features through a simple API. Tasks create children tasks by calling the following inlined, non-blocking function:

```
espresso::enqueue<type>(taskFn,
         [timestamp, locale,] args...)
```

The new task will run the task function `taskFn` with arguments supplied through registers. If `timestamp` or `locale` are given, the task will be synchronized according to their semantics.

Espresso programs start by creating one or more initial tasks with `espresso::enqueue` and calling `espresso::run`, which returns control after all tasks finish. Listing 5.1 shows the implementation of `sssp` described in Section 5.1.1, which we will use to explain Espresso's semantics. In this example, the program creates one initial task to visit the source vertex.

5.2.1 Espresso Semantics

Espresso runs all speculative tasks atomically, i.e., speculative tasks never appear to interleave. Moreover, Espresso provides strong atomicity [53, 96] between speculative and non-speculative tasks: the effects of a speculative task are invisible to non-speculative tasks until the speculative task commits (*containment* [53]), and non-speculative writes do not appear mid-speculative-task (*non-interference* [53]). Espresso does not guarantee atomicity among non-speculative tasks.

Atomicity has implications on the allowed concurrency between parent and child tasks. If a parent creates a speculative child, the child appears to execute after the parent finishes. If a speculative parent creates a non-speculative child, the child does not run until after the parent commits (e.g., Section 5.1.2).

There are no atomicity guarantees among non-speculative tasks. The programmer

Task mode	**Synchronization mechanism**	
	Timestamps	Locales
Non-speculative	barriers	mutual exclusion
Speculative	ordered commits	reduce conflicts

Table 5.1: The effect of Espresso's synchronization mechanisms.

must ensure non-SPEC tasks are *well-synchronized*, that is, they avoid race conditions.

Espresso provides two synchronization mechanisms to control how the system executes tasks. *Timestamps* enforce order among tasks, and *locales* enforce mutual exclusion. Timestamps and locales have consistent semantics for both speculative and non-speculative tasks, but have different effects on concurrency, described below and summarized in Table 5.1.

Timestamps: Like with Swarm (Chapter 3), timestamps are integers that specify a partial order among tasks. If two tasks have distinct timestamps, the system ensures they appear to execute in timestamp order. A timestamped task may only assign timestamps greater than or equal to its own to its children. A non-timestamped task cannot create timestamped children.

Timestamps impose *barrier semantics* among non-speculative tasks. For example, a non-speculative task with timestamp 10 will not run until all tasks with timestamp < 10 have finished and committed. However, speculative tasks can run out of order, speculating past these barriers. Timestamps only *constrain the commit order* of speculative tasks. Hence, speculation can increase parallelism for ordered algorithms.

Listing 5.1 shows timestamps in action. Each `sssp` task has a timestamp that corresponds to its path's distance to the source vertex. If all tasks had type `NONSPEC` instead of `MAYSPEC`, this code would implement the non-speculative, one-distance-at-a-time `sssp` version from Section 5.1.1. If all tasks had type `SPEC`, the code would implement the fully speculative `sssp` version.

Locales: A locale is an integer that, if specified, denotes the data the task will access. Locales enforce mutual exclusion: if two tasks have the same locale, Espresso guarantees that they do not run concurrently. For tasks that only need to acquire a single lock, locales are a more efficient alternative to conventional shared-memory locks. Moreover, Espresso hardware uses locales to map tasks that are likely to access the same data to the same chip tile in order to exploit locality, as we saw with spatial hints (Chapter 4).

For non-speculative tasks, locales can be used as mutexes to write safe parallel code. For speculative tasks, locales are not necessary, since these tasks already appear to execute atomically. Locales are still useful in reducing aborts (Chapter 4) as well as exploiting locality across speculative and non-speculative tasks.

Listing 5.1 shows locales in action. Each `sssp` task uses the ID of the vertex it processes as its locale. For non-speculative tasks, this implements mutual exclusion among tasks that access the same vertex. For both speculative and non-speculative tasks, this approach sends all tasks that operate on the same vertex to the same chip tile, improving temporal locality. To avoid cache ping-ponging, we apply the optimization from Chapter 4 that uses the cache line of the vertex as the locale.

These synchronization mechanisms cover important use cases, but are not exhaustive. For example, locales only provide single-lock semantics, but a task may need to acquire multiple locks. In this case, the task may either use shared-memory locks or resort to speculative execution by marking the task SPEC. Espresso does not support multiple locales per task because doing so would be much more complex.

Comparison with other execution models: Espresso generalizes Swarm, HTM, and message-driven processors. Swarm programs consist of all-timestamped SPEC tasks. Espresso extends Swarm to support non-speculative tasks and to make timestamps optional. Locales extend and restrict spatial hints to provide mutual exclusion among non-speculative tasks. Espresso also subsumes HTM. HTM programs consist of transactional (speculative) and non-transactional (non-speculative) code blocks. These are equivalent to non-timestamped SPEC and NONSPEC tasks. Finally, Espresso can emulate message-driven processors [98, 276], supporting an actor model [10], through locale-addressed, non-timestamped NONSPEC tasks.

5.2.2 MAYSPEC: Tasks That May Speculate

Espresso tasks run speculatively or not. However, the programmer must choose one of three types for each task: SPEC, NONSPEC, or MAYSPEC. MAYSPEC lets the system decide which mode the task should run in. This is useful as there are times when it is safe to run a task speculatively but not non-speculatively. If the system wants to dispatch a MAYSPEC task that cannot yet run non-speculatively, the task runs speculatively.

Choosing between NONSPEC and MAYSPEC affects performance but not correctness. NONSPEC and MAYSPEC tasks must already be well-synchronized, so they are also safe to run speculatively. If the task will be expensive to run speculatively (e.g., the logging tasks in Section 5.1.2), it should be NONSPEC. Otherwise, MAYSPEC lets the system decide.

Listing 5.1 shows MAYSPEC in action. All `sssp` tasks are tagged as MAYSPEC because they can run in either mode, as locales enforce mutual exclusion among same-vertex tasks. This implements the right strategy discussed in Section 5.1.1: tasks with the lowest unprocessed distance run non-speculatively, and if this non-speculative parallelism is insufficient, the system runs higher-distance (i.e., higher-timestamp) tasks speculatively.

5.2.3 Exception Model

Espresso does not restrict the actions that tasks may perform. Beyond accessing shared memory, both speculative and non-speculative tasks may invoke *irrevocable actions* that cannot be undone by a versioned memory system. That is, tasks in either running mode can call into arbitrary code, including code that triggers exceptions and invokes system calls.

To provide precise exception semantics and enforce strong atomicity, a speculative task that triggers an exception or a system call yields until it becomes the earliest active task and is then *promoted* to run non-speculatively. To guarantee promoted tasks still appear strongly atomic, a promoted task does not run concurrently with any other non-speculative task (Section 5.4.1). Previous TLS systems used similar techniques to provide precise exceptions [168, 348].

Promotions can be expensive but are rare in practice. To avoid frequent and expensive promotions, tasks that frequently invoke irrevocable actions should use the `NONSPEC` type. Capsules further reduce the need for promotions.

Finally, Espresso introduces a `promote` instruction to expose this mechanism. If called from a speculative task, `promote` triggers an exception that will, in the absence of conflicts, eventually promote the task. If called from a non-speculative task, `promote` has no effect. `promote` has two uses. First, it can be invoked by tasks that detect an inconsistency and know they must abort, similar to transactional retry [172]. Second, `promote` lets code that must perform an expensive action avoid doing so speculatively (e.g., Listing 5.2 in Section 5.3).

5.3 Capsules

Although hardware version management is more efficient than a software-only equivalent [67, 115], hardware-only speculation can cause more serialization. For example, Espresso supports irrevocable actions in speculative tasks by promoting them to run non-speculatively, an expensive process that limits parallelism. Irrevocable actions cannot run under the control of hardware speculation, since hardware cannot undo their effects. This is limiting, because letting speculative tasks invoke system calls in parallel has many legitimate uses [39], like concurrent reads from disk. Beyond system calls, tasks may wish to perform software-managed speculative actions that exploit application-specific parallelization strategies, such as commutativity [86, 218, 268, 313]. As we saw in Section 5.1.3, prior work proposed escape actions to achieve this goal [64, 264, 411]. But escape actions are incompatible with systems for ordered parallelism that need speculative forwarding. Forwarding can make speculative tasks lose data and control-flow integrity, making it impossible for software to dependably undo speculative actions [163].

Specifically, escape actions suffer from two problems with forwarding. First, a mispeculating task that has lost control-flow integrity may jump to malformed or invalid code that initiates an escape action and performs an unintended system call, such as overwriting a file or exiting the program, that cannot be undone. Second, mispeculating tasks may clobber state used by escape actions, causing them to misbehave when they read this uncommitted data. For example, consider an escape action that allocates memory from a free list. A mispeculating task can temporarily clobber the free list, causing the escape action to return invalid data or crash.

To address these issues, we present *Capsules*, a technique to enable safe software-managed speculative actions in any speculative system. Capsules are a powerful tool for systems programmers. Similar to escape actions, Capsules can avoid the overheads of hardware conflict detection, perform irrevocable actions, and undo speculative actions by registering abort handlers. Capsules enable programmers to guarantee safety even if a mispeculating task attempts to use Capsules incorrectly. It does this through two mechanisms. First, it provides *untracked memory* that is protected from mispeculating tasks. Second, it uses a *vectored-call interface* that guarantees control-flow integrity within a capsule. We add three new instructions to the ISA, `capsule_call`, `capsule_ret`, and `capsule_abort_handler`. We explain their semantics below.

5.3.1 Untracked Memory

We allow memory segments or pages in the application's address space to be classified as *untracked*. Untracked memory is neither conflict-checked nor versioned in hardware, eliminating speculation overheads for accesses to untracked data. We use standard virtual memory protection mechanisms to prevent speculative tasks from accessing untracked memory without entering a capsule (Section 5.4.2). This is analogous to how OS kernel memory is protected from userspace code.

Software-managed speculative state should be maintained in untracked memory both to avoid the overhead of hardware conflict detection as well as to ensure it is not corrupted by speculative tasks. Accesses to untracked data can be synchronized conventionally (e.g., with locks) to ensure safety.

5.3.2 Safely Entering a Capsule

Since a speculative task can lose control-flow integrity, we need a way to enter a capsule that guarantees the integrity of capsule code. To achieve this, we use a *vectored-call* interface, similar to that of system calls.

We require that all capsule code is wrapped into *capsule functions* placed in untracked memory. A *capsule-call vector* stored in untracked memory contains pointers to all capsule functions. Since speculative tasks cannot access untracked memory, they can only call capsule functions with the `capsule_call` instruction. `capsule_call` is similar to an ordinary `call` instruction, but it takes an index into the capsule-call vector as an operand instead of a function address. `capsule_call` looks up the index in the vector. If the index is within bounds, it jumps to its corresponding function and disables hardware speculation; if the index is out of bounds, it triggers an exception. `capsule_ret` is used to return from a capsule function.

The vectored-call interface retains safety even when speculative tasks lose control-flow integrity. A task can only enter a capsule through a `capsule_call` instruction, which can only jump to the beginning of a known capsule function.

5.3.3 Capsule Execution

A capsule may access untracked memory and perform irrevocable actions such as system calls without triggering a promotion. It typically operates on untracked memory, but may also access tracked memory (e.g., to make data such as file contents available to non-capsule speculative tasks). Its accesses to tracked memory use the normal conflict detection and resolution mechanisms. This ensures loads from tracked memory return valid data if the capsule is running non-speculatively, and that the enclosing task will eventually abort if a capsule reads invalid data while running speculatively.

Like a system call, a capsule function cannot trust its caller to be well behaved, as the caller could be mispeculating. A speculatively running capsule may receive invalid data through arguments or tracked memory, or perhaps should not have been called due to control mispeculation. To handle these, it may register an abort handler to compensate for its actions. It uses the `capsule_abort_handler` instruction, which takes a function pointer and arguments as operands. The given function will run non-speculatively if the capsule's enclosing task aborts.

A capsule function running speculatively must ensure it only performs actions for which it can safely compensate. It must check its arguments and data read from tracked memory before using the data in an unsafe way. To avoid performing rare actions that would be very expensive or unsafe to perform speculatively, it may use the `promote` instruction, which is a no-op if running non-speculatively, but causes the enclosing task to abort if it was speculative and immediately exits the capsule. Thus, code following a `promote` instruction will only run non-speculatively, is guaranteed to be in a consistent state, and any abort handlers it registers will not run.

5.3.4 Capsule Programming Example

Listing 5.2 shows how `malloc` can be written as a capsule function. `malloc` first checks that its stack pointer is valid and has sufficient space, using `promote` otherwise. `malloc`

```
void* malloc(size_t bytes) {
  if (BAD_STACK()) promote;
  if (bytes > (16 << 20)) promote;
  if (bytes == 0) capsule_ret(nullptr);
  void* ptr = do_alloc(bytes);
  capsule_abort_handler(&do_dealloc, ptr);
  capsule_ret(ptr);
}
```

Listing 5.2: `malloc` implemented as a capsule function.

also checks whether the requested allocation is very large, using `promote` if the program wants to allocate more than 16 MB. This avoids wasting excessive space to satisfy large requests from mispeculating tasks. After these checks, `malloc` calls `do_alloc`, which allocates the requested chunk. Finally, `malloc` uses `capsule_abort_handler` to register a call to `do_dealloc` as the abort handler. In this example, `do_alloc` and `do_dealloc` are thread-safe functions that use conventional synchronization (e.g., locks) to perform allocation and deallocation of heap memory. All allocator metadata (e.g., free lists) are stored in untracked memory. If the calling task aborts, the call to `do_dealloc` runs, freeing the allocated memory.

5.4 Implementation

We implement Espresso and Capsules by extending Swarm (Chapter 3). Swarm is a strong baseline because it already provides most of the mechanisms needed for Espresso: it efficiently supports fine-grain tasks, implements scalable ordered speculation using timestamps, and performs locality-aware execution (Chapter 4). However, Espresso could be implemented over classic TLS systems as well, and Capsules are a general technique that could be applied to any speculative system, including HTM, TLS, Swarm, or Espresso. We first describe how Swarm's features are extended to implement Espresso, then describe the implementation of Capsules.

5.4.1 Espresso Microarchitecture

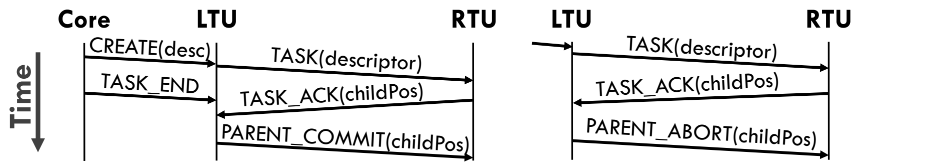
Espresso generalizes the Swarm microarchitecture to *(i)* support non-speculative tasks, *(ii)* handle their interactions with speculative tasks, and *(iii)* implement exceptions.

Tasks use the same hardware task descriptor format as in Swarm, with two additional bits to store the type (`SPEC`, `NONSPEC`, or `MAYSPEC`). The dispatch, queuing, speculation mechanisms, and commit protocol of `SPEC` tasks are unchanged from those of Swarm in Section 3.3.

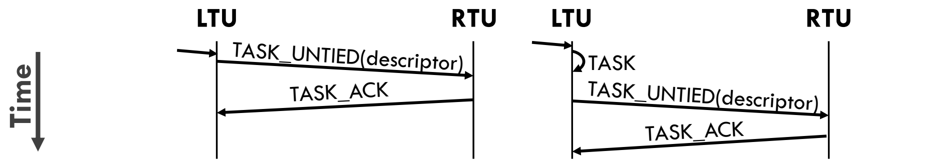
Non-speculative tasks require simple changes to the dispatch and queuing logic in every tile's task unit. A `NONSPEC` task may run only when *(i)* its parent is non-speculative or committed, *(ii)* it is not timestamped or its timestamp matches that of the earliest active task, *(iii)* it has no locale or its locale does not match that of any running task, and *(iv)* the system is not satisfying a promotion.

The tile's dispatch logic performs all these checks using local state. It picks the lowest-timestamp task available to run, but excludes `NONSPEC` tasks that are not yet runnable. Locales regulate task dispatch in the same way as spatial hints (Section 4.2): tasks with the same locale are enqueued to the same tile and serialized, providing mutual exclusion.

A non-speculatively running task frees its task queue entry when dispatched and does not use a commit queue entry. This reduces queue pressure. Recall that when



(a) Speculative parent creates speculative child, ends, then commits (b) Speculative parent creates speculative child, then aborts



(c) Non-speculative parent creates child (d) Speculative parent keeps non-speculative child locally until commit

Figure 5-4: Swarm (a,b) and Espresso (a,b,c,d) enqueue protocol between a local task unit (LTU) and a remote task unit (RTU), as a parent task creates a child.

a speculative task creates children, the local task unit asynchronously sends their task descriptors to remote tiles and tracks their destinations to enable parent commit or abort notifications (Section 3.3.2), as shown in Figure 5-4(a) and Figure 5-4(b). In contrast, since a non-speculative task can never abort, it does not track its children or send them any notifications, as shown in Figure 5-4(c). This reduces traffic and allows non-speculative tasks to create an unbounded number of children; their children can always be spilled to memory (Section 3.3.7).

Mixing non-speculative and speculative tasks requires changing the dispatch, conflict detection, and commit mechanisms:

1. Speculative `NONSPEC` *enqueue:* If a speculative task creates a `NONSPEC` child destined to a remote tile, since the child cannot run before its parent commits, the child task descriptor is buffered at its parent's local task unit, and the request to enqueue the child is sent to the remote tile only once the parent commits (Figure 5-4(d)). This avoids needless abort and commit notifications.

2. `MAYSPEC` *dispatch:* A `MAYSPEC` task is always speculatively runnable, but may exploit non-speculative execution for efficiency. The dispatch logic checks if the task meets the same conditions for a `NONSPEC` task to run. If so, the task executes non-speculatively; otherwise, it executes speculatively.

3. Conflicts: A non-speculative task does not track read/write-sets. However, to implement strong atomicity, its accesses are conflict-checked against speculative tasks. A non-speculative access that conflicts with a *running* speculative task's read/write set aborts the speculative task.

If a non-speculative task N conflicts with a *finished* speculative task S, S may be unsafe to abort. Recall that tiles periodically communicate to find the virtual time (VT) of the earliest active task, and all finished tasks whose VTs precede that earliest active VT must then commit (Section 3.3.6). The *global virtual time arbiter* tracks the earliest active VT in the system. If *S*'s tile has sent a locally earliest active VT to the commit arbiter that is higher than *S*'s VT, *S* may later be deemed committed by the arbiter. To handle this race, *S*'s tile NACKs *N*'s conflicting memory request, causing *N* to stall until the arbiter replies and *S*'s fate is known. This race is very rare.

4. Commit protocol: When tiles send their earliest active VT to the arbiter, the timestamps of non-speculative tasks are included for consideration. This prevents any speculative task with a higher timestamp from committing, while allowing same-timestamp speculative tasks to commit. As they inherently "win" all conflicts, non-speculative tasks have no tiebreaker.

Exceptions: Any attempt by a speculative task to perform an irrevocable action (e.g., a system call or segmentation fault) causes a *speculative exception*. There are two causes for speculative exceptions: either the task legitimately needs to execute an irrevocable action, or it is a mispeculating task performing incorrect execution.

Whereas TLS schemes stall the core running the exceptioned task [168,348], Espresso leverages commit queues to avoid holding up a core. The *exceptioned* task is immediately stopped and its core becomes available to run another task. Its writes are rolled back, and its children tasks are aborted and discarded. Espresso then keeps the task's *read set* active in the commit queue. If the read set detects a conflict with an earlier task, the exceptioned task was mispeculating, so it becomes runnable again for *speculative* execution. However, if the task becomes the earliest active task without having suffered a conflict, it legitimately needs to perform an irrevocable action.

After an exceptioned task's tile finds it became the earliest active task in the system, the tile promotes it to re-run non-speculatively. This proceeds as follows. First, the task's tile sends a promotion request to the virtual time arbiter. The arbiter forbids other tiles from dispatching further non-speculative tasks. This is because the promoted task was speculative, so it must run isolated from all other tasks. After all currently running non-speculative tasks have finished, the exceptioned task is promoted and allowed to run. Although the promoted task cannot run concurrently with other non-speculative tasks (to maintain isolation), other speculative tasks can continue execution, ordered after the promoted task. Though expensive, this process happens rarely.

In summary, Espresso requires simple extensions to Swarm, and in return substantially improves performance and programmability, as we will see in Section 5.5.

5.4.2 Capsules Implementation

Capsules extend the system to implement untracked memory and a vectored-call interface to capsule functions.

Untracked memory: Our implementation of untracked memory makes simple extensions to standard virtual memory protection mechanisms. In addition to the standard read, write, and execute permissions bits, we add an *untracked permission bit* to each page table entry and TLB entry. An access to an untracked page from a speculative task that is not in a capsule causes a memory-protection exception. Programs can request tracked or untracked memory using the `mmap` system call, and change a page's permissions with the `mprotect` system call. Alternatively, untracked memory could be implemented as a new virtual memory segment.

To track whether the current task is in a capsule, each core has a *capsule depth* counter, initialized to zero at task start to indicate that the task is not yet in a capsule. `capsule_call` increments the capsule depth counter by one, and `capsule_ret` decrements it by one. This allows capsule functions to call other capsule functions, while tracking when the task finally exits the outermost capsule.

Safely entering a capsule: The *capsule-call vector* contains pointers to all capsule functions and is stored at a fixed location in untracked memory. It would be cumbersome to manually assign unique IDs to capsule functions and build the call vector. However, the linker and loader can automate this process, similarly to how they handle position-independent code [193].

Capsule aborts: If a task aborts, any registered abort handlers must run non-speculatively. The task creates each abort handler as a `NONSPEC` task with no timestamp: unordered with respect to timestamped tasks. In slight contrast to a regular `NONSPEC` child of a speculative task, the abort handler task descriptor is buffered at its parent's local tile, then remotely enqueued once its parent *aborts*.

Special handling is required to abort a task while it is still executing a capsule. Normally, a task is immediately stopped after detecting a conflict. A capsule, however, cannot be stopped arbitrarily—it must be allowed to complete, and then its abort handlers run, to guarantee a consistent state in untracked memory. Nonetheless, to avoid priority inversion, i.e., letting an aborting task block the execution of an earlier-ordered task, our implementation always handles conflicts immediately upon detection. If the task is in a capsule when it needs to be aborted, all the task's side-effects to tracked memory are rolled back immediately, so other tasks can proceed to use the recovered state in tracked memory. Abort notifications are sent to its children. The capsule is then marked as *doomed* and allowed to continue execution. A doomed capsule's writes to tracked memory are not performed, nor are its enqueues. Its accesses to untracked memory are performed normally. The core becomes available to run another task after the doomed capsule exits.

5.5 Evaluation

We evaluate Espresso and Capsules on a diverse set of applications. Non-speculative execution brings modest performance and efficiency gains when non-speculative parallelism is plentiful, but forcing non-speculative execution with `NONSPEC` can dramatically hurt parallelism. By contrast, `MAYSPEC` achieves the best of both worlds and can be applied indiscriminately without hurting parallelism, making it easy to use. Capsules yields order-of-magnitude speedups in important use cases, which we show through

Application	**Input**	**1-core cycles**		**1-core perf.**
		total	**per task**	**vs. serial**
sssp [289]	cage14 [100]	1.6 B	53	0.93×
	East USA roads [1]	2.4 B	299	1.74×
cf [338, 407]	movielens-1m [170]	1.5 B	59500	0.98×
triangle [338]	R-MAT [72]	59.5 B	1240	1.02×
kmeans [257]	m40 n40 n16384 d24 c16	8.6 B	6500	1.02×
color [174]	netflix [42]	11.1 B	163	1.42×
bfs [230]	hugetric-00020 [26, 100]	3.3 B	139	0.93×
mis [339]	R-MAT [72]	1.7 B	121	0.80×
astar (Chapter 3)	Germany roads [280]	1.6 B	458	1.37×
genome [257]	g4096 s48 n1048576	2.3 B	850	1.01×
des [289]	csaArray32	1.7 B	506	1.82×
nocsim [7]	16x16 mesh, tornado	19.3 B	979	1.79×
silo [371]	TPC-C, 4 whs, 1 Ktxns	0.1 B	3380	1.13×

Table 5.2: Benchmarks: source implementations and inputs; run time, average task length, and serial-relative performance on a single-core system.

two case studies on memory allocation and disk-based key-value stores.

5.5.1 Methodology

Like in Section 4.3.1 we model systems of up to 256 cores (Figure 4-2), with the same parameters (Table 4.2) and scaling methodology.

Benchmarks: Table 5.2 reports the benchmarks and inputs used to evaluate Espresso and the Capsules-based allocator. We consider 13 ordered and unordered benchmarks. We ported 10 benchmarks from Chapter 3 and Chapter 4. Seven of the 10 benchmarks have tasks that can be well-synchronized with timestamps and locales: `sssp`, `color`, `bfs`, `astar`, and `des` are ordered applications, and `genome` and `kmeans` are unordered transactional applications. We introduce `mis`, adapted from PBBS [339]. It finds the maximal independent set of an unweighted graph: 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. We implement an ordered and deterministic variant. Chapter 6 describes `mis` and its Swarm implementation in more detail.

We also introduce bulk-synchronous `cf` (collaborative filtering) and `triangle` (triangle counting), ported from Ligra [338, 407]. Their tasks are well-synchronized: they perform lock-free atomic updates and use barriers, which we replace with timestamps.

Section 5.5.2 compares Espresso versions by declaring all well-synchronized tasks as `SPEC`, `NONSPEC`, or `MAYSPEC`. The source code is otherwise identical. Swarm runs all tasks speculatively, and is thus equivalent to Espresso's `SPEC`. We also evaluate state-of-the-art software-only parallel versions as in the previous chapters. (except for `genome` and `kmeans`, which lack a non-transactional parallel version, and `astar`, for which software-parallel versions yield no speedup).

Four of the benchmarks have a significant amount of dynamic memory allocation: genome, des, nocsim, and silo. Section 5.5.3 compares the effect of different allocators on the SPEC (Swarm) version of these applications.

We report speedups relative to *tuned* 1-core Swarm implementations. These are *coarse-grain* variants, when applicable, which have less overhead but are less scalable than their fine-grain counterparts (Section 4.4). Due to hardware task management, 1-core Swarm versions are competitive with (and often faster than) tuned software-only *serial* implementations, as shown in Table 5.2. Like in Section 3.4, we report results for the full parallel region.

Memory allocation: Only two of the benchmarks used in Section 5.5.2 (genome and des) allocate memory within tasks. To separate concerns, we study the impact of allocators in Section 5.5.3 and use an ideal memory allocator in Section 5.5.2. As described in Section 3.4, the ideal allocator is implemented within the simulator, and allocates and deallocates heap memory from per-core pools with zero overhead. Memory freed by a speculative task is not reused until the task commits. For fairness, software-only implementations also use this allocator.

5.5.2 Espresso Evaluation

Figure 5-5 compares the performance of Swarm (SPEC), Espresso's NONSPEC and MAYSPEC variants, and the software-only parallel versions as the system scales from 1 to 256 cores. Because most applications are hard to parallelize, MAYSPEC always matches or outperforms the software-only versions, which scale poorly in all cases except sssp-cage, cf, triangle, and color. Thus, we do not consider software-only versions further. Among the other schemes, Swarm works poorly on cf and triangle, and sacrifices some performance on sssp-cage, genome, and color; NONSPEC scales well in sssp-cage, cf, triangle, genome, kmeans, and bfs, but performs poorly in other applications because it forgoes opportunities to exploit speculative parallelism; and MAYSPEC always performs best.

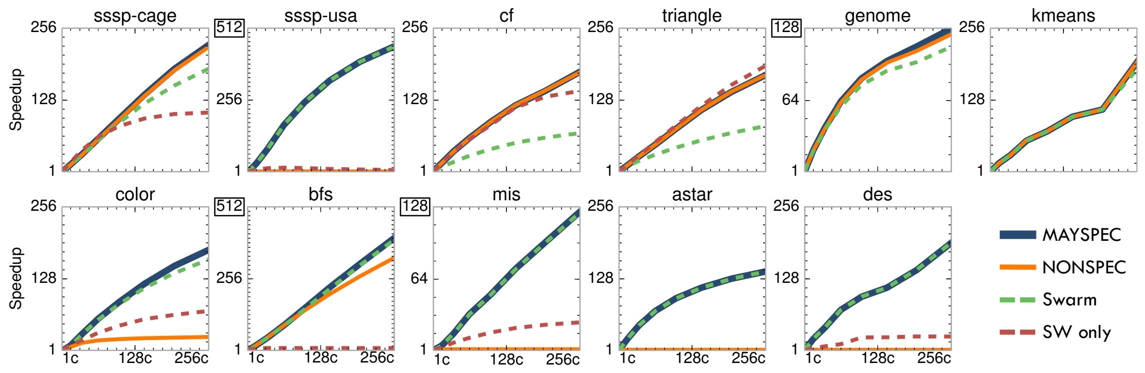
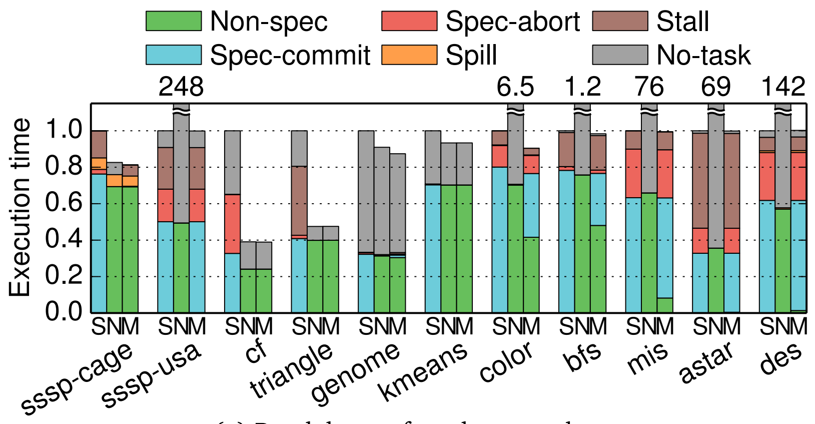
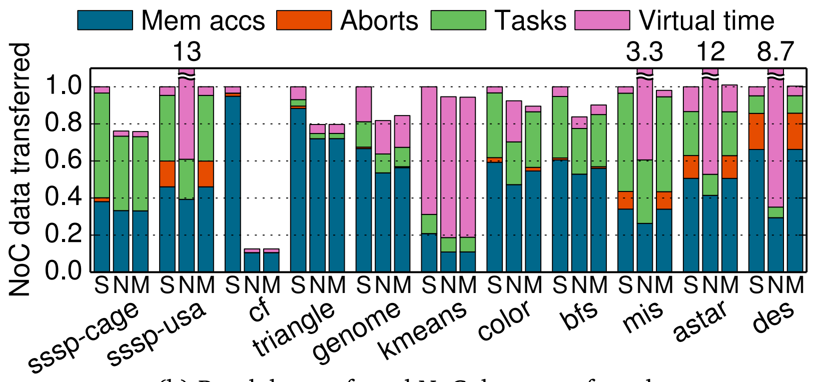


Figure 5-5: Speedup of Swarm (SPEC), NONSPEC, MAYSPEC, and software-only benchmark variants on 1–256 cores. Higher is better.



(a) Breakdown of total core cycles



(b) Breakdown of total NoC data transferred

Figure 5-6: Breakdowns at 256 cores, using Swarm (SPEC), and Espresso's NONSPEC, and MAYSPEC variants. Each bar is normalized to Swarm. Lower is better.

Figure 5-6 gives more insight into these results by showing core cycle and network traffic breakdowns at 256 cores for the Swarm, NONSPEC, and MAYSPEC versions. Each group of bars shows breakdowns for a different application. The height of a bar in Figure 5-6a is the execution time relative to Swarm. Each bar shows a breakdown of how cores spend these cycles, executing *(i)* non-speculative tasks, or *(ii)* speculative tasks that later commit or *(iii)* later abort; *(iv)* spilling tasks to memory; *(v)* stalled on a full task or commit queue; or *(vi)* idle because there are no tasks available to run. Each bar of Figure 5-6b reports the total bytes injected into the NoC relative to Swarm, broken down into four categories: *(i)* memory accesses from running tasks (between L2s and L3, or L3 and main memory), *(ii)* abort traffic (parent abort notifications and rollback memory accesses), *(iii)* task enqueues and parent commit notifications, *(iv)* virtual time updates (for ordered commits and barriers).

For sssp, as discussed in Section 5.1.1, neither Swarm nor NONSPEC perform best across inputs. MAYSPEC outperforms the best of Swarm and NONSPEC by using speculation opportunistically. In the shallow graph (cage), MAYSPEC runs almost all tasks non-speculatively. Meanwhile, in the deep graph (usa), MAYSPEC runs almost all tasks speculatively, overlapping the processing of vertices at multiple distances to extract enough parallelism. NONSPEC and MAYSPEC spend fewer cycles executing tasks than Swarm's committed cycles because non-speculative execution is more efficient: it reduces cache pressure (no undo log) and network traffic (less cache pressure, no aborts, and no parent commit notifications).

cf and triangle show the largest difference between Swarm and Espresso variants. Both applications have plentiful non-speculative parallelism, but some tasks are large. When tasks run speculatively in cf they fill their Bloom filters and yield false conflicts, whereas in triangle, the long tasks prevent short tasks from committing, leading to full queues. NONSPEC and MAYSPEC are up to 2.6× (cf) faster than Swarm, and have up to 8.0× (cf) lower network traffic.

genome, kmeans, color, and bfs show similar trends as sssp-cage. genome has a phase with little parallelism; non-speculative execution runs faster in this phase, reducing no-task stalls. Though STAMP's kmeans is nominally transactional, locales and timestamps non-speculatively synchronize it, so NONSPEC and MAYSPEC perform equally. Nearly all traffic is virtual time updates because locales effectively localize accesses to shared data, resulting in a high L2 hit rate.

The final three benchmarks show similar trends as sssp-usa: mis, astar, des have little non-speculative parallelism, even though nearly all their tasks can be safely declared NONSPEC. Therefore, NONSPEC performance is terrible, up to 142× worse than Swarm (des). NONSPEC is dominated by no-task stalls as only a few same-timestamp tasks run at a time. MAYSPEC addresses this pathology and matches Swarm.

These results show that Espresso both improves performance and efficiency while *also aiding programmability*. Across the 11 results, NONSPEC achieves 29× gmean speedup at 256 cores, Swarm (SPEC) 162×, and MAYSPEC scales to 198×. Without MAYSPEC, programmers would need to know how much non-speculative parallelism is available to decide whether to use NONSPEC or SPEC. MAYSPEC lets them declare any task that may run non-speculatively as such without performance concerns.

5.5.3 Capsules Case Study: Dynamic Memory Allocation

We design a scalable speculation-friendly memory allocator using Capsules. As discussed in Section 5.1.3, simply calling memory allocation routines within speculative tasks introduces needless conflicts that limit parallelism. Prior work has proposed allocators for software TM [194], studied existing allocators' synchronization impact on software TM [33], and used HTM to accelerate allocators *for non-speculative parallel programs* [108, 117, 224]. TMs without forwarding can avoid false conflicts on allocator metadata by using escape actions or open-nested transactions [411], but as far as we know, no memory allocator has been implemented for systems with speculative forwarding.

To this end, we implement capalloc, a memory allocator built with Capsules. All allocation calls (malloc, calloc, etc.) are capsule functions, implemented following the pattern in Listing 5.2. To avoid reusing memory freed by tasks that later abort, each

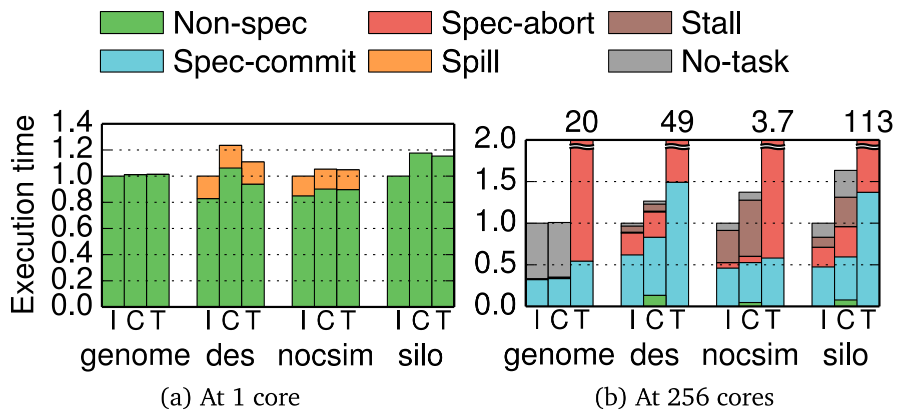


Figure 5-7: Normalized execution time using three implementations of dynamic memory allocation: Ideal, Capalloc, and unmodified TCMalloc. Lower is better.

deallocation call (`free`, `cfree`) creates a `NONSPEC` child with no timestamp to perform the deallocation. Thus, memory is deallocated only after the caller commits.

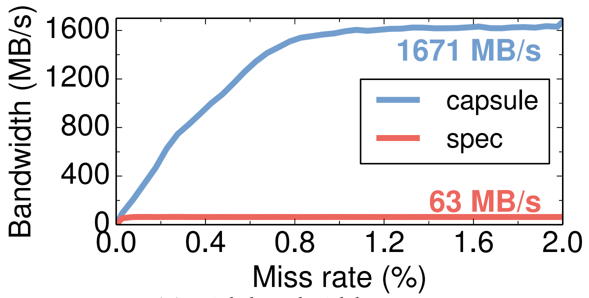
`capalloc`'s internal design mimics TCMalloc [154], a state-of-the-art and widely used memory allocator. Small allocations (≤16 KB in our implementation) are served by a per-core software cache. These caches hold a limited amount of memory, and allocate from a set of central freelists. Large allocations are served from a centralized page heap that prioritizes space efficiency. The central freelists, large heap, and system-wide page allocator use spinlocks to avoid data races.

The key difference between `capalloc` and TCMalloc is that `capalloc` keeps all its metadata in untracked memory. TCMalloc implements freelists as linked lists using the free chunks themselves. The free chunks cannot be placed in untracked memory as they are used by speculative tasks.

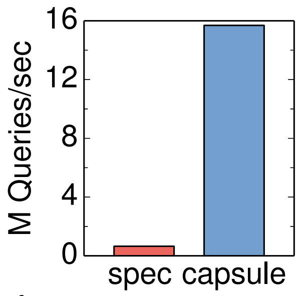
As explained in Section 5.5.1, we evaluate `capalloc` on the four applications with frequent dynamic allocation. We compare `capalloc` with TCMalloc and the ideal allocator.

Figure 5-7a shows single-core results, which let us examine work efficiency without concern for parallelism. Each group of bars shows execution times for one application, normalized to the ideal allocator. `capalloc` and TCMalloc perform similarly, adding gmean slowdowns of 11% and 8%, respectively.

Figure 5-7b shows 256-core results. TCMalloc suffers spurious conflicts among tasks that access the same allocator metadata, and is gmean 25× slower than the ideal allocator. By contrast, `capalloc` is only gmean 30% slower than the ideal allocator. These overheads are in line with those in the single-core system, demonstrating `capalloc`'s scalability. `capalloc` is gmean 20× faster than TCMalloc—from 3× (`nocsim`) to 69× (`silo`).



(a) Disk bandwidth usage



(b) Performance at 2% miss rate

Figure 5-8: Disk utilization of `spec` and `capsule` variants of a key-value store.

5.5.4 Capsules Case Study: Disk-Backed Key-Value Store

The previous case study showed that Capsules avoid needless conflicts; we now show the benefits of letting speculative tasks perform controlled parallel I/O. We implement a simple disk-backed key-value store that runs the YCSB [91] benchmark with 4 KB tuples and 80/20% read/write queries. Each query runs within a single speculative task. The key-value store keeps only some of the tuples in main memory. If the requested tuple is not in main memory, it must be fetched from disk and another tuple must be evicted, writing it back to disk if it is dirty.

We implement two miss-handling strategies. First, `spec` performs the disk fetch and eviction directly in `SPEC` tasks, without Capsules. Because this requires read (and possibly write) system calls, tasks that suffer a miss are promoted and run serially. Second, `capsule` performs each fetch from a capsule function invoked within the `SPEC` task, and performs each eviction from a follow-up `NONSPEC` task. This lets `capsule` perform parallel I/O.

We evaluate both strategies on a 256-core system with an NVMe SSD.[4] Figure 5-8a shows how disk bandwidth grows with miss rate (which we control by varying the memory footprint). `spec` tops out at 63 MB/s, far below the disk's bandwidth, due to its serialized I/O. By contrast, `capsule` fully saturates disk bandwidth, achieving 1671 MB/s. Figure 5-8b shows that, with a 2% miss rate (where both variants are I/O-bound), `capsule` achieves 24× the throughput of `spec`. These results show that concurrent system calls can be highly beneficial, and Capsules successfully unlock this benefit for speculative tasks.

5.6 Additional Related Work

Espresso is most closely related to Swarm, but draws from prior HTM and TLS systems as well. Table 5.3 summarizes the capabilities of these systems.

[4] We model a Samsung 960 PRO, which supports 440K/360K IOPS for random 4 KB reads/writes, with minimum latencies of 70/20 μs [322].

5.6.1 Task Scheduling and Synchronization

Prior work has investigated hardware support for scheduling and synchronization of *either* speculative or non-speculative tasks. On the speculative side, prior techniques avoid aborts on known dependences by stalling [405] or pipelining [376], enable threads to speculate past barriers [169, 248, 327], and accelerate multi-word compare-and-swap in hardware transactions [243]. On the non-speculative side, prior work has proposed hardware-accelerated task-stealing [221, 326] and dataflow [68, 98, 129, 161, 276] schedulers. The lack of shared synchronization mechanism hinders HTM, where mixing transactional and conventional synchronization is unsafe [119, 382]. Prior work has crafted software primitives that bypass transactional mechanisms [119, 382], toggle between transactional and lock-based synchronization [315], or move data structures from kernel to user space [390].

By contrast, Espresso's timestamps and locales facilitate coordination *across* speculative and non-speculative tasks. This opens the door to `MAYSPEC`, which allows the system to dynamically choose to execute tasks speculatively or non-speculatively. Moreover, timestamps and locales offer more performance for non-speculative tasks than shared-memory barriers and locks. Timestamps are essentially hardware-accelerated barriers [41, 210, 330]. Locales are handled by the task dispatch logic, so they are more efficient than hardware-accelerated locks [208, 241, 410], as they eliminate spinning within a task. Locales also enable locality-aware task mapping.

5.6.2 Restricted vs. Unrestricted Speculative Tasks

TLS systems are unrestricted: their tasks can run arbitrary code, although only the earliest active task may run a system call or exception handler. Most HTMs are restricted: they forbid transactions from invoking irrevocable actions, which hinders programmability. OneTM [52] and TCC [169] permit unrestricted transactions. Our promotion technique lies between OneTM-serialized, which pauses all other threads, and OneTM-concurrent, which keeps all other threads running but requires in-memory metadata to support unbounded read/write sets. By contrast, Espresso keeps only speculative tasks

Capability	**HTM**	**TLS**	**Swarm**	**Espresso**
Ordered parallelism	~[a]	✓	✓	✓
Non-speculative parallelism	✓	✗	✗	✓
Shared synchronization mechanisms	✗	✗	✗	✓
Locality-aware	✗	✗	✓	✓
Unrestricted speculative code	~[b]	✓	✗	✓

[a] Most HTMs are unordered (Section 2.4.3).
[b] Most HTMs are restricted (Section 5.6.2).

Table 5.3: Comparison of prior systems and Espresso.

running through a promotion. TCC, like TLS, does not support non-speculative parallelism (all code runs speculatively except the transaction with commit permission).

5.6.3 Open-Nested Transactions

Some speculative tasks must perform operations that would be expensive or incompatible with their hardware speculation mechanisms. Escape actions (Section 5.1.3) are one prior solution for HTMs, as are open-nested transactions [251, 264, 266], which run within another transaction and commit immediately after finishing, before its enclosing transaction commits. Like Capsules, open-nested transactions still use ordinary conflict detection to preserve atomicity when accessing data shared by other transactions. Like escape actions and Capsules, open-nested transactions use abort handlers to undo their effects. Unfortunately, open-nested transactions are also unsafe with speculative forwarding because open-nested transactions may lose data and control-flow integrity and then perform harmful writes and commit.

5.7 Summary

We have presented two techniques that bring the benefits of non-speculative parallelism to systems with ordered speculation, like Swarm from Chapter 3. First, the Espresso execution model efficiently supports speculative and non-speculative tasks, provides shared synchronization mechanisms to all tasks, and lets the system adaptively run tasks speculatively or non-speculatively to achieve the best of both worlds. Second, Capsules let speculative tasks safely invoke software-managed speculative actions, bypassing hardware version management and conflict detection. We have shown that these techniques improve performance and enable new capabilities, such as scaling memory allocation and allowing speculative tasks to safely perform parallel I/O.

CHAPTER 6

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

This work was conducted in collaboration with Suvinay Subramanian, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. The Fractal execution model was developed collaboratively. This thesis contributes the database and MIS motivation studies. This thesis also contributes to the development of the high-level interface, applications, and the architectural simulator.

Systems for speculative parallelism, such as hardware transactional memory (HTM), thread-level speculation (TLS), or Swarm (Chapter 3), 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-grain 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 (Section 2.4.3) 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-grain 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 [169,344] or small Bloom filters [69,399]. But these tracking structures have limited capacity and force transactions that overflow them to serialize, even when they have no conflicts [52, 169, 229, 399]. 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.

Through its efficient support for programmer-controlled task order, we have seen in Chapter 3 and Chapter 4 that the Swarm architecture can address some of these problems. By exposing timestamps to software, Swarm can be used to 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-grain 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 6.1). 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, as evaluated in Chapter 3. This exploits intra-transaction parallelism, and, at 256 cores, it is 21× faster than running operations within each transaction serially (Section 6.1.2). 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.

This chapter presents two main contributions that achieve these goals. The first contribution is Fractal, an execution model for nested speculative parallelism that generalizes Swarm. 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.

The second contribution is a simple implementation of Fractal that builds on Swarm and supports arbitrary nesting levels cheaply (Section 6.3). 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 [28, 375], 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 [27, 28]. The Fractal execution model lets our implementation avoid these problems. Beyond exploiting more parallelism, focusing on fine-grain tasks reduces the hardware costs of speculative execution.

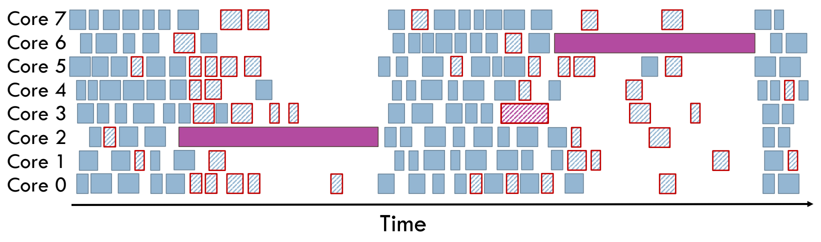
This chapter demonstrates Fractal's performance and programmability benefits through several case studies (Section 6.1) and a broad evaluation (Section 6.4). Fractal uncovers abundant fine-grain parallelism on large programs. For example, ports of the STAMP benchmark suite to Fractal outperform baseline 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.

6.1 Motivation

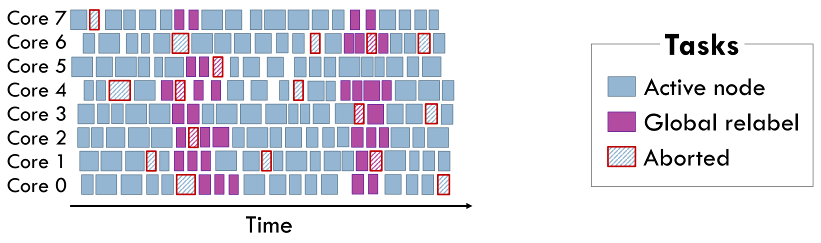
We motivate Fractal through three case studies that highlight its key benefits: uncovering abundant parallelism, improving programmability, and avoiding over-serialization. Since Fractal subsumes prior speculative execution models (HTM, TLS, and Swarm), all case studies use the Fractal architecture (Section 6.3), 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.

6.1.1 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 [81], but it is hard to parallelize [40, 289]. 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



(a) `maxflow-flat`



(b) `maxflow-fractal`

Figure 6-1: 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.

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 [81] 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 [18].

Since push-relabel can process active nodes in an arbitrary order, it can be parallelized using transactional tasks of two types [289,304]. An active-node task operates on a node and its neighbors, and may enqueue other tasks to process newly-activated nodes. A global-relabel task performs a global relabel operation. Every 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 6.4.1 for methodology details.) At 256 cores, `maxflow-flat` scales to 4.9× only. Figure 6-1a 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, as we saw in the previous chapters, 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

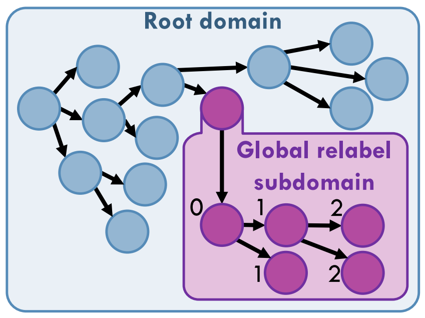


Figure 6-2: In `maxflow-fractal`, each global-relabel task creates an ordered subdomain.

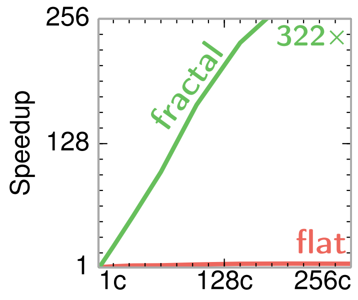


Figure 6-3: Speedup of different `maxflow` versions on 1–256 cores.

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-grain ordered tasks, as shown in Figure 6-2. A global-relabel task and its subdomain appear as a single atomic unit with respect to other tasks in the (unordered) root domain. Figure 6-1b illustrates how this improves parallelism and efficiency. As a result, Figure 6-3 shows that **`maxflow-fractal`** achieves a speedup of 322× at 256 cores (over **`maxflow-flat`** on one core).

Fractal is the first architecture that effectively exploits maxflow's fine-grain nested parallelism: neither HTM, nor TLS, nor Swarm can support the combination of unordered and ordered parallelism latent in maxflow. Prior software-parallel push-relabel algorithms attempted to exploit this fine-grain parallelism [18, 289, 304], 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` [40] and Galois [289]. On 1–256 cores, they achieve maximum speedups of only 4.9× and 8.3× over `maxflow-flat` at one core, respectively.

6.1.2 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 from this chapter's introduction. Conventional HTMs run each database transaction in a single thread, and exploit coarse-

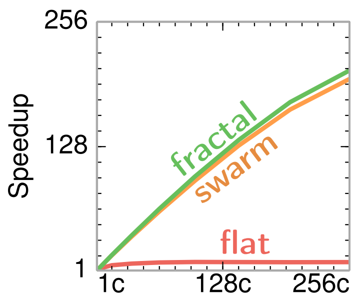


Figure 6-4: Speedup of `silo` versions on 1–256 cores.

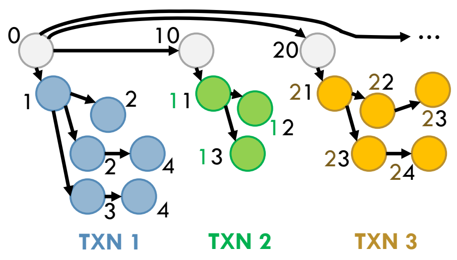


Figure 6-5: `silo` Swarm uses disjoint timestamp ranges for different database transactions, sacrificing composability.

grain inter-transaction parallelism only. But amid its queries and updates, 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 6-2. We apply both approaches to the `silo` in-memory database [371]. Figure 6-4 shows that, at 256 cores, **`silo-fractal`** scales to 206×, while **`silo-flat`** scales to 9.7× only, 21× slower than **`silo-fractal`**.

Figure 6-4 also shows that **`silo-swarm`**, the Swarm version of `silo` from Chapter 3 and Chapter 4, achieves similar performance to **`silo-fractal`** (**`silo-swarm`** is 4.5% slower). Figure 6-5 illustrates `silo-swarm`'s implementation: the transaction-launching code assigns disjoint timestamp ranges to transactions (10 contiguous timestamps per transaction in Figure 6-5), and each transaction enqueues tasks only within this range (e.g., 10–19 for TXN 2 in Figure 6-5). `silo-swarm` uses the same fine-grain tasks as `silo-fractal`, exposing plentiful parallelism and reducing the penalty of conflicts (Section 3.5.1). For example, in Figure 6-5, 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-grain parallelism because each nested transaction merges its speculative state with that of its parent [27, 28]. We compare Fractal and parallel nesting HTMs in detail in Section 6.5, after discussing Fractal's implementation. Beyond these issues, parallel nesting HTMs do not support ordered parallelism, so they would not help `maxflow` or `silo`.

6.1.3 Fractal Avoids Over-Serialization

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

Consider the maximal independent set (mis) problem which, given a graph, seeks 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.

A straightforward, non-deterministic mis algorithm uses unordered, atomic tasks [339]. 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 independent set and marking its neighbors as excluded from the set. mis-flat creates one task for every node in the graph, and finishes when all these tasks have executed. Figure 6-6 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 6-6 shows that **mis-fractal** scales to 145× at 256 cores, 48% faster than **mis-flat**.

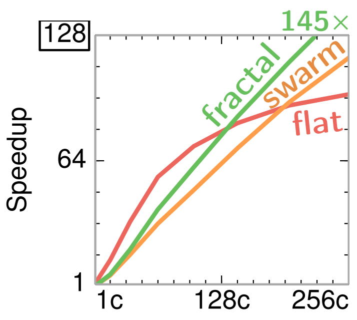


Figure 6-6: Speedup of different mis versions on 1–256 cores.

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 6-6 shows that **mis-swarm** scales to 117×, 24% slower than **mis-fractal**, as unnecessary order constraints cause more aborted work.[1]

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.

[1] mis-swarm's order constraints make it deterministic, which some users find desirable [46, 47, 106].

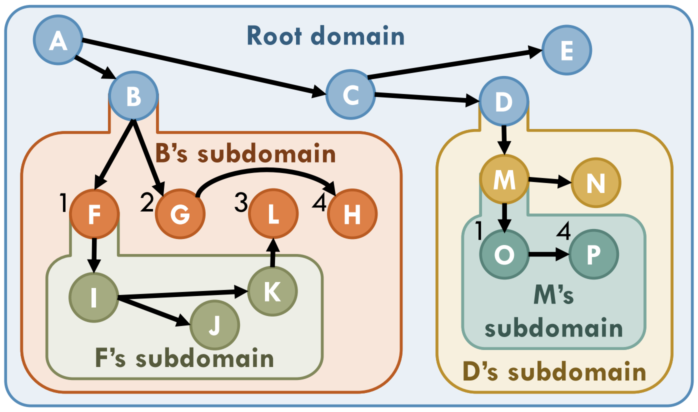


Figure 6-7: Elements of the Fractal execution model. Arrows point from parent to child tasks. Parents enqueue their children into ordered domains where tasks have timestamps, such as A's and M's subdomains, or unordered domains, such as the other three domains.

6.2 Fractal Execution Model

Fractal programs consist of *tasks* in a logical hierarchy of nested *domains*. Each task may access arbitrary data, and may create child tasks as it finds new work to do. Figure 6-7 illustrates the key elements of the Fractal execution model. For example, in the figure, task C creates children D and E. When each task is created, it is enqueued to a specific domain.

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., every child is ordered after its parent. For example, in Figure 6-7, 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: all tasks execute atomically and in isolation, as though executed serially and new tasks are scheduled in an unordered bag.

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 level-by-level breadth-first search in `maxflow-fractal` (Section 6.1.1). For example, in Figure 6-7, the timestamps of tasks F, G, L, and H ensure they appear to run in that fixed order. These semantics are the same as those of Swarm (Chapter 3).

Semantics across domains: Each task can create a single subdomain and enqueue tasks into it. For example, in Figure 6-7, 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 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, 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.

A task may also enqueue child tasks to its immediate enclosing domain, or *superdomain*. For example, in Figure 6-7, 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.

6.2.1 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 6.1 illustrates the key features of the low-level Fractal interface by showing the implementation of the `mis-fractal` tasks described in Section 6.1.3. Similar to Swarm and Espresso, 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

```
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 6.1: Fractal implementation of `mis` tasks.

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 [356]. 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 6.1: High-level interface functions.

task's timestamp. This isn't the case in Listing 6.1, as this non-deterministic `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 6.1, 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 (Chapter 4). 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.

Fractal is also orthogonal to Espresso's support for speculative and non-speculative tasks, and Capsules' ability to safely bypass hardware speculation (Chapter 5). We leave combining these techniques to future work.

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 [29]. Table 6.1 details its main constructs, and Listing 6.2 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 ac-

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

Listing 6.2: Pseudocode for Fractal implementation of `mis`'s `include` using the high-level interface.

tual syntax is slightly more complicated because we do not modify the compiler, and we implement these constructs using macros.[2]

6.3 Fractal Implementation

Our Fractal implementation seeks three desirable properties. First, the architecture should perform *fine-grain 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 6.5). Second, *creating a domain should be cheap*, as domains with few tasks are common (e.g., `mis` in Section 6.1.3). 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.

6.3.1 Fractal Virtual Time

Recall from Section 3.3.4 that Swarm maintains a consistent total order among tasks by assigning 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 6-8. 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.

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

[2] The difference between the pseudocode in Listing 6.2 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.

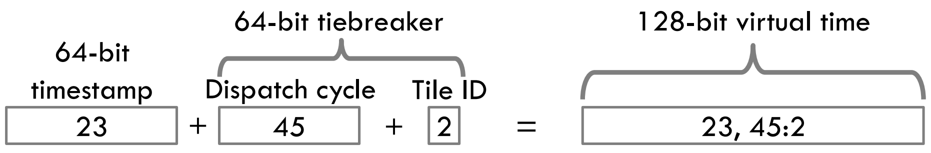


Figure 6-8: Swarm VT construction.

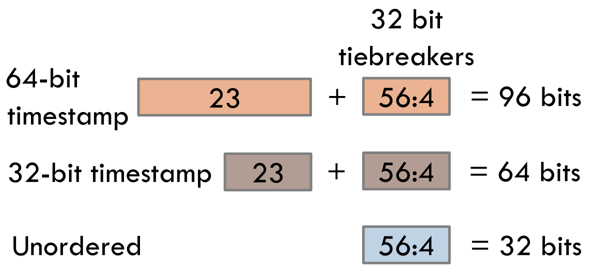


Figure 6-9: Domain VT formats.

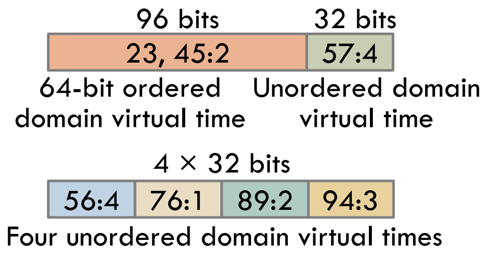

96 bits
32 bits
23, 45:2
57:4
64-bit ordered domain virtual time
Unordered domain virtual time
4 × 32 bits
56:4
76:1
89:2
94:3
Four unordered domain virtual times

Figure 6-10: Example 128-bit fractal VTs.

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 parent before child. While 32-bit tiebreakers are efficient, they can wrap around. Section 6.3.3 discusses how Fractal handles wrap-arounds. Figure 6-9 illustrates the possible formats of a domain VT, which can take 32, 64, or 96 bits.

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 6-10 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 6.3.2 describes how to support levels beyond those that can be represented in 128 bits.

Figure 6-11 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

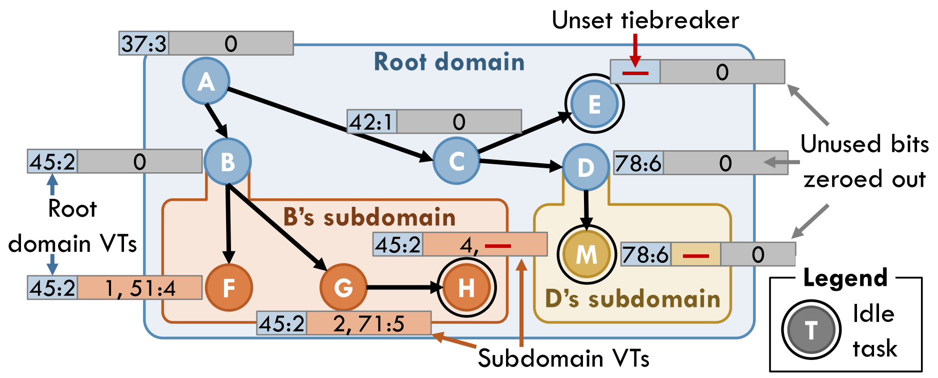


Figure 6-11: Fractal VTs in action.

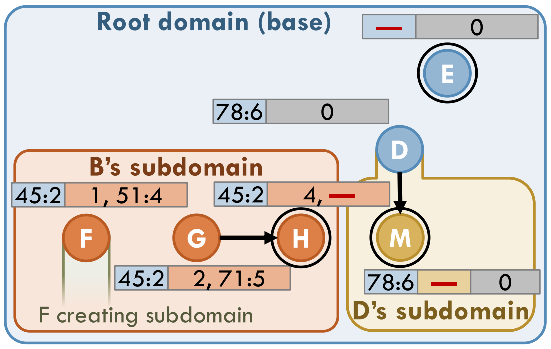
dispatched tasks can be ordered by comparing their fractal VTs. For example, F (in B's subdomain) is ordered after B, but before M (in D's subdomain). Fractal performs fine-grain speculation by using the total order among running tasks to commit and resolve conflicts at the level of individual tasks. 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 6-11, 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.

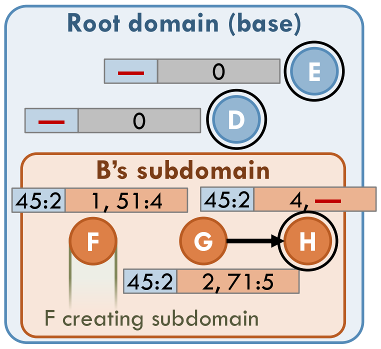
In summary, fractal VTs capture all the information needed for ordering and task enqueues, so these operations do not rely on centralized structures. Moreover, the rules of fractal VT construction automatically enforce Fractal's semantics across domains while performing speculation only at the level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

6.3.2 Supporting Unbounded Nesting

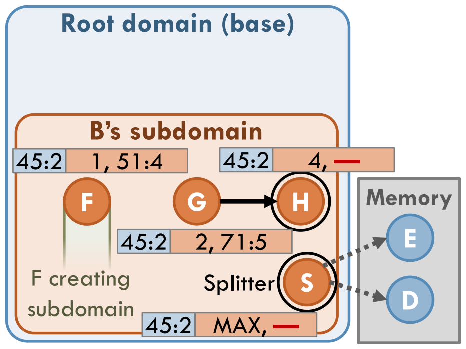
Large applications may consist of parallel algorithms nested with arbitrary depth. Fractal supports this unbounded nesting depth by spilling tasks from shallower domains to memory. These spilled tasks are filled back into the system after deeper domains finish. This process, which we call *zooming*, is conceptually similar to the stack spill-fill mechanism in architectures with register windows [166]. Zooming in allows Fractal to continue fine-grain speculation among tasks in deeper domains, without requiring additional structures to track speculative state. Note that, although zooming is in-



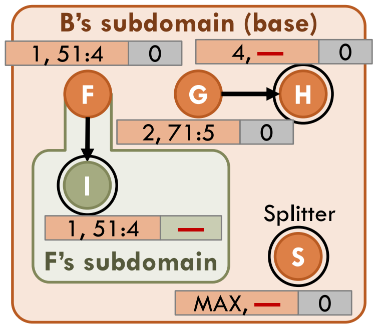
(a) After B commits.



(b) Base-domain tasks abort.



(c) Base-domain tasks are spilled.



(d) B's subdomain becomes the base domain.

Figure 6-12: Starting from Figure 6-11, zooming in allows F to create and enqueue to a subdomain by shifting fractal VTs.

volved, it imposes negligible overheads: zooming is not needed in our full applications (which use two nesting levels), and it happens infrequently in microbenchmarks (Section 6.4.4).

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 6-11, 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 global virtual time arbiter (Section 3.3.6) with its fractal VT.

Figure 6-12 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 6-12a shows the state of the system 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 6-12a, 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 (Section 3.3.5). Figure 6-12b 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 (Section 3.3.7): task units dispatch *coalescer* tasks that remove base-domain tasks from task queues, store them in memory, and enqueue *splitter* tasks that will later re-enqueue the spilled tasks. The splitter task is deprioritized relative to all regular tasks in the domain. Figure 6-12c 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 (Section 3.3.4). 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 6-12d 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* attempts to enqueue to its superdomain. Such an action is treated as an exception (Section 5.2.3), so the enqueuing task waits (off core) until all tasks preceding it have committed. Once non-speculative, 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 [362] 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.

6.3.3 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 process exploits the property that, if the task that created a domain precedes all other active tasks (i.e., is non-speculative), 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.

6.3.4 Putting It All Together

Our Fractal implementation adds small hardware overheads over Swarm.[3] 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_task` 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.

[3] Swarm itself imposes modest overheads to implement speculative execution (Section 3.3.8)

	Application	Input	1-core run time (B cycles)
Ch. 3,4,5	**color** [174]	com-youtube [233]	0.968
	mis [339]	R-MAT [72], 8 M nodes, 168 M edges	1.34
	msf [339]	kron_g500-logn16 [26, 100]	0.717
	silo [371]	TPC-C, 4 whs, 32 Ktxns	2.98
STAMP [257]	**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 [40]	rmf-wide [81, 156], 65 K nodes, 314 K edges	16.7

Table 6.2: Benchmark information: source implementations, inputs, and execution time on a single-core system.

6.4 Evaluation

We now analyze the benefits of Fractal in depth. As in Section 6.1, we begin with applications where Fractal uncovers abundant fine-grain 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.

6.4.1 Methodology

Like in Section 4.3.1 we model systems of up to 256 cores (Figure 4-2), with the same parameters (Table 4.2) and scaling methodology. For task scheduling, we use spatial hints with load balancing (Section 4.5). We use 128-bit fractal VTs and simulate 2 cycles per `create_subdomain` instruction.

Benchmarks: Table 6.2 reports the benchmarks we use for evaluation. Benchmarks have 1-core run-times of about 1 B cycles or longer. We use four benchmarks from Chapter 3, Chapter 4, and Chapter 5, which we adapt to Fractal; Fractal implementations of the eight STAMP benchmarks [257]; and one new Fractal workload, `maxflow`, which we adapt from `prsn` [40].

Benchmarks adapted from the previous chapters use their same inputs. `msf` includes an optimization to filter out non-spanning edges efficiently [46]. This optimization improves absolute performance but reduces the amount of highly parallel work, so `msf` has lower scalability than the unoptimized Swarm version of Chapter 3.

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 [156], one of the harder graph

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

families from the DIMACS maxflow challenge [40].

Like in Section 3.4, we 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.

6.4.2 Fractal Uncovers Abundant Parallelism

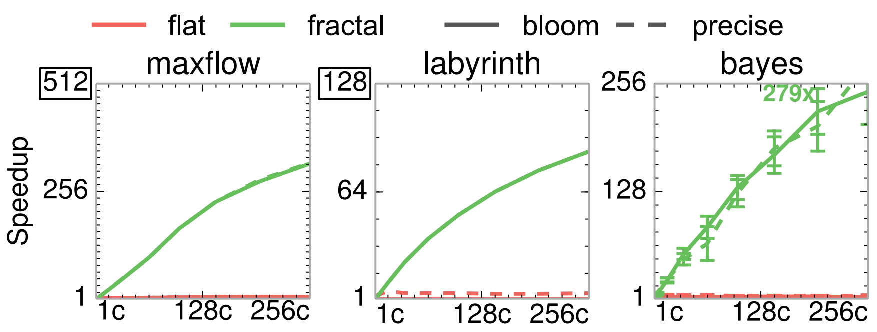
Fractal's support for nested parallelism greatly benefits three benchmarks: `maxflow`, as well as `labyrinth` and `bayes`, the two least scalable benchmarks from STAMP.

maxflow, as discussed in Section 6.1.1, is limited by long global-relabel tasks. Our `fractal` version performs the breadth-first search nested within each global relabel in parallel.

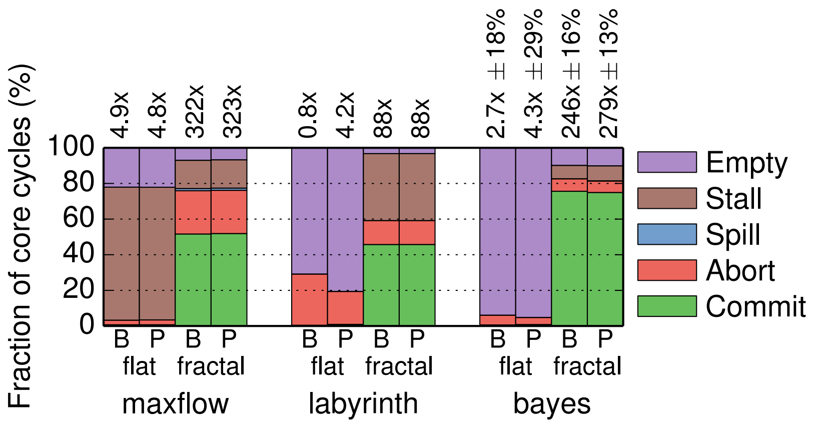
labyrinth finds non-overlapping paths between pairs of *(start, end)* cells on a 3D grid. Each transaction operates on one pair: it finds the shortest path on the grid and claims the cells on the path for itself. In the STAMP implementation, each transaction performs this shortest-path search sequentially. 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 6.3 compares the 1-core performance and average task lengths of `flat` and `fractal` versions. `flat` versions of these benchmarks have long, unordered trans-



(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 6-13: Performance of `flat` and `fractal` versions of applications with abundant nested parallelism, using Bloom filter–based or Precise conflict detection.

actions (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 [69, 258, 325, 399].

Figure 6-13a 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].

[4] Note that systems with more tiles have higher cache and queue capacities, which sometimes cause

Figure 6-13b gives more insight into these differences by showing the percentage of cycles that cores spend on different activities: *(i)* running tasks that are ultimately 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 group of bars shows results for a different application at 256 cores.

Figure 6-13b 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-grain nested speculative parallelism is an effective way to scale challenging applications.

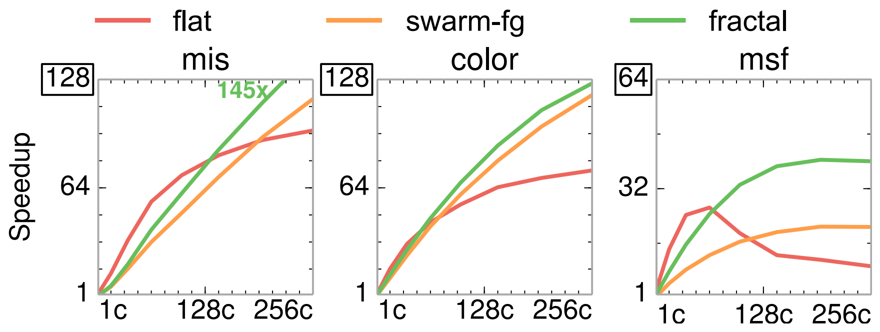
6.4.3 Fractal Avoids Over-Serialization

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-grain operations or groups of tasks (Section 6.1.3). Section 6.1 showed that this has a negligible effect on `silo`, so we focus on the other three applications.

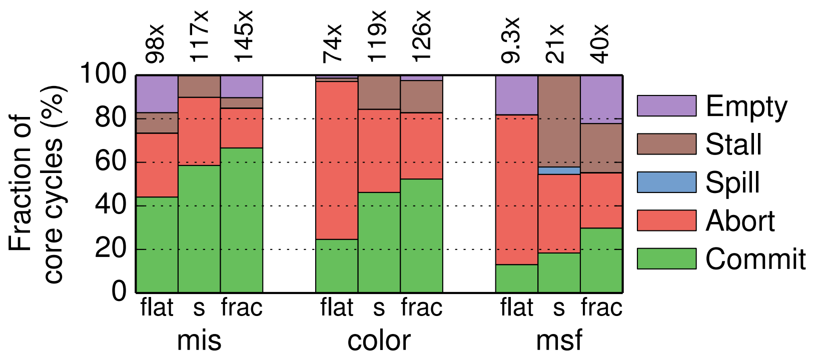
`mis`, `color`, and `msf` are graph-processing applications. Their `flat` versions perform operations on multiple graph nodes that can be parallelized but must remain atomic—e.g., in `mis`, adding a node to the independent set and excluding its neighbors (Section 6.1.3). `mis-flat` is unordered, while `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-grain operation to exploit this nested parallelism (Table 6.3). The `swarm-fg` versions of these benchmarks use the same fine-grain tasks as `fractal` but use a unique timestamp or timestamp range per coarse-grain operation to guarantee atomicity, imposing a *fixed* order among coarse-grain operations.

Figure 6-14 shows the scalability and cycle breakdowns for these benchmarks. `flat` versions achieve the lowest speedups, from 26× (`msf` at 64 cores) to 98× (`mis`). Figure 6-14b 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.

superlinear speedups (Section 4.3.1)



(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 6-14: 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.

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 6.3).

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 6-14b 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-grain parallelism more attractive by avoiding needless order constraints.

6.4.4 Zooming Overheads

Although our Fractal implementation supports unbounded nesting (Section 6.3.2), two nesting levels suffice for all the benchmarks we evaluate. Larger programs should re-

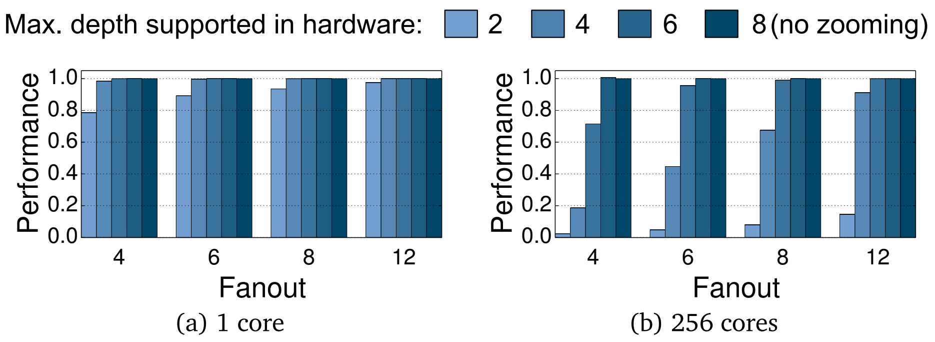


Figure 6-15: Characterization of zooming overheads.

quire 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 6-15a 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 6-15b 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 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.

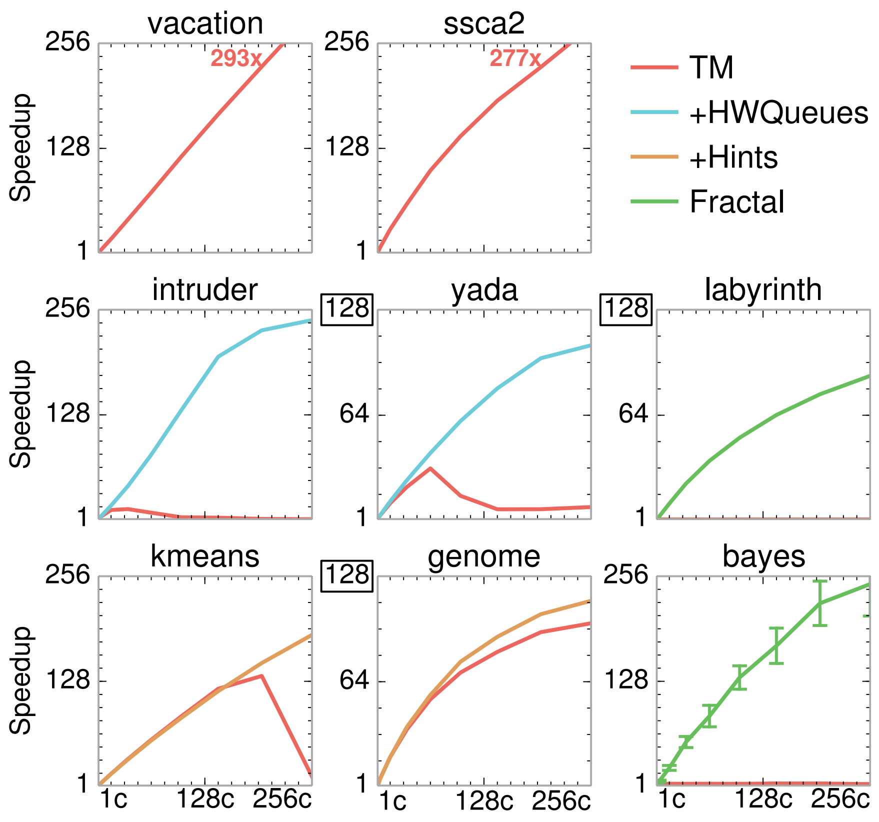


Figure 6-16: Different Fractal features make all STAMP applications scale well to 256 cores.

6.4.5 Discussion

We considered 20 benchmarks to evaluate Fractal: all eight from STAMP [257], the remaining 11 applications from the previous chapters, as well as `maxflow`. We looked for opportunities to exploit nested parallelism, focusing on benchmarks with limited speedups. In summary, Fractal benefits 7 out of these 20 benchmarks. We did not find opportunities to exploit nested parallelism in the seven Swarm and Espresso benchmarks not presented here (`bfs`, `sssp`, `astar`, `des`, `nocsim`, `cf`, and `triangle`). These benchmarks all scale well to 256 cores, the latter two have plentiful non-speculative parallelism, and the former five already use sufficiently fine-grain tasks.

Figure 6-16 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 (Section 3.3.2) makes them scale. Second, spatial hints (Chapter 4) improves `genome` and makes `kmeans` scale. Finally, as we saw in Section 6.4.2, 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.

6.5 Additional Related Work

6.5.1 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* [264]: 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 [17,55,83,84] and make conflicts more likely.

Prior work has explored relaxed nesting semantics, like open nesting [251,264, 272] and early release [341], 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 [267]: 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 [12,27,109, 383], but these suffer from even higher overheads than flat STMs. Vachharajani [375, Ch. 7] and FaNTM [28] 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-grain 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 [28]. Workarounds for these issues are complex and sacrifice performance (e.g., a nested transaction eventually aborts all its ancestors for livelock avoidance [28]). 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.

6.5.2 Nesting in Thread-Level Speculation

Renau et al. [309] use timestamps to allow out-of-order task spawn, exploiting nested parallelism across function calls and loop nests [238]. Each task carries a timestamp range, and splits it in half when it spawns a successor. 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.

6.5.3 Nesting in Non-Speculative Systems

Nesting is supported by most parallel programming languages, such as OpenMP [121]. In many languages, such as NESL [48], Cilk [142], and X10 [76], 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 [79] or dataflow annotations [120], to avoid data races. Though nested non-speculative parallelism is often sufficient, many algorithms need speculation to be parallelized efficiently [289]. By making nested speculative parallelism practical, Fractal brings the benefits of composability and fine-grain parallelism to a broader set of programs.

6.6 Summary

We have presented Fractal, a new execution model for fine-grain 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-grain 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× at 256 cores.

CHAPTER 7

Conclusion

Transistor density scaling has slowed and voltage scaling has long tapered out, so improved performance and energy efficiency will only come from more effective use of modern multi-billion-transistor chips. This demands simple expression and efficient extraction of all types of application parallelism, to the point of executing thousands of operations at a time, or more. Unfortunately, the interfaces of current multicores, GPUs, and accelerators makes this an arduous job, if not impossible, as they favor easier types of parallelism or coarse-grain tasks. Prior hardware architectures to simplify parallelization do not scale sufficiently to reach our goal. Exploiting parallelism has for too long been restricted to a minority of expert programmers and application domains.

This thesis has presented a versatile execution model, architecture, and cross-layer techniques that enable more programmers to exploit some of the most challenging types of parallelism from a broader range of applications. In particular, we have made the following contributions:

- **Swarm** (Chapter 3) is a co-designed execution model and multicore architecture with ordered tasks at the software-hardware interface. Swarm programs express potential parallelism through dynamically created timestamped tasks. Timestamps make synchronization implicit, enabling the programmer to specify the unique, few, or many task execution orders that correctly satisfy their application semantics. The Swarm microarchitecture extracts fine-grain ordered irregular parallelism by running tasks speculatively and out of order, adding modest changes to a conventional multicore. It innovates on prior speculative architectures through its hardware task management for short, dynamic tasks; decentralized and scalable speculation mechanisms that enable speculative forwarding with selective aborts; and its hierarchical protocol that achieves high-throughput ordered commits.

- **Spatial hints** (Chapter 4) extends the Swarm execution model and microarchitecture to perform locality-aware execution, pushing the scalability of the system to hundreds of cores. The programmer conveys their knowledge of locality by optionally tagging each task with an integer hint that denotes the data it is likely to access. The hardware maps tasks with the same hint to the same tile to localize data accesses, and balances load by moving hints, and their tasks, around the system. We showed that, by exploiting Swarm's support for tiny ordered tasks, most data accessed is at least abstractly known just before a task is created, and programs can often be restructured to further isolate accesses to contentious data.
- **Espresso** and **Capsules** (Chapter 5) improve the efficiency of the Swarm architecture by combining the benefits of non-speculative parallelism with the simplicity and scalability of Swarm's speculative parallelism. Espresso extends Swarm to support non-speculative tasks, provides timestamps and locales as common coordination mechanisms for tasks running in either mode, and introduces the `MAYSPEC` task type to let the system decide whether to run individual tasks speculatively or not. Capsules let speculative tasks safely invoke software-managed speculative actions, even in the face of speculative forwarding. By bypassing hardware version management and conflict detection, this enables new capabilities, such as scalable memory allocation and file I/O from speculative tasks.
- **Fractal** (Chapter 6) pushes the generality of the system by seamlessly composing nested speculative parallelism, while building on Swarm's support for speculative ordered parallelism. Fractal expresses computation as a hierarchy of nested domains, where any task may create an ordered or unordered subdomain and enqueue its children into it. Tasks within a domain appear to execute in its timestamp order, and as one atomic unit to the tasks outside the domain.

This thesis implemented the ordered-tasks execution model atop a baseline homogeneous single-threaded multicore. This enabled us to explore how to efficiently extract fine-grain ordered irregular parallelism with modest changes to a now-ubiquitous architecture. Beyond multicores, our prior work has also adapted these techniques to multithreaded architectures [5]. We therefore expect that the underlying principles of this thesis can be applied to make challenging types of parallelism practical in other throughput-oriented architectures like GPUs and heterogeneous and specialized architectures.

7.1 Future Work

Our contributions open exciting research avenues spanning computer architecture, computer systems, software, and compilers.

Fine-grain parallelism at large scale: Distributed-memory systems become increasingly enticing as Moore's Law nears its end. For example, near-data processing architectures [31] improve energy efficiency by reducing data movement, and large-scale systems that span multiple chips [209, 232], boards, or racks [36] improve performance by scaling an application across thousands to millions of cores. Beyond shared-memory parallel architectures, could the techniques in this thesis democratize access to these systems by efficiently supporting fine-grain irregular parallelism? Unfortunately, Swarm and other speculative architectures leverage the coherence protocol to detect conflicting accesses cheaply. While coherence protocols can scale to high core counts [247], their added complexity and overhead may make them undesirable at very large scales. Fortunately, accurate spatial hints and locales could help make global coherence unnecessary: if all accesses to shared read-write data are localized within a tile, all conflicts become local, and coherence becomes superfluous across tiles. We have shown that hints effectively localize the vast majority of read-write accesses. Fractal further simplifies this pursuit, allowing each remote memory access to execute as a separate task, sent to the appropriate tile, all while retaining the atomicity of the original task. This would reuse speculation logic to emulate coherence.

Exploiting task choice to adaptively limit misspeculation: Although speculation can unlock abundant irregular parallelism, we should avoid as many aborts as possible. Swarm's large speculation window with many more tasks than cores gives the architecture *choice* on when, where, and how to execute tasks. Spatial hints exploit that choice by spatially mapping tasks to reduce data movement and serializing same-hint tasks, while Espresso decides whether to execute MAYSPEC tasks speculatively or not. Can we exploit task dispatch choice to further reduce aborts? For instance, task predictors could estimate a dispatch candidate's likelihood to commit, factoring in several signals (e.g., function pointer address, arguments, and history). Should a tile make local decisions to select the most profitable task to dispatch, or should there be a global approach to throttle concurrency, as in our prior work on speculation-aware multithreading [5]? Pursuing these questions can build on the abundance of prior work in branch prediction, TM schedulers [19, 44, 45, 402], adaptive speculative discrete-event simulation [350], and control speculation in the original TLS [344].

Expressing more types of parallelism: This thesis presented an execution model whose hardware implementation extracts abundant fine-grain irregular parallelism across a large suite of applications. However, it is likely not a panacea: we will need new techniques to express and efficiently extract parallelism from challenging applications that goes beyond dynamic hierarchical timestamp-ordered tasks. First, algorithms may exhibit more complex task scheduling patterns. For instance, suppose the programmer knows that, in their program, many tasks create new work scheduled far away in program order. In Swarm or Fractal, the new task descriptors are queued in hardware, but may later spill to and from memory using our general-purpose mechanisms. However, these actions could be optimized with application-specific memory-backed schedulers. What is right interface that lets programmers control these task overflow structures, and what hardware support should be provided? Second, timestamps are insufficiently expressive to order the tasks of some algorithms and forego optimization opportunities. For example, applications like `sssp` can tolerate reordering of tasks but heuristically order them for algorithmic efficiency [253, 271]. Could we exploit this fact by generalizing task order with timestamp *intervals*? Tasks with disjoint intervals would be appropriately ordered yet those with overlapping intervals are respectively unordered. Or should the system decouple timestamped dispatch order from unordered commits? In either case, what are the architectural implications, and can we improve efficiency with more flexible task orders?

Application to new algorithms and domains: This thesis demonstrated that the proposed techniques can extract abundant parallelism from conventionally hard-to-parallelize algorithms across several domains. We hope this motivates future work that applies fine-grain parallelism to a broader range of algorithms. For instance, Fractal's composable timestamps can implement algorithms that rely on non-monotone priority queues [365], but we have yet to explore this direction. This could unlock scalable implementations of Prim's algorithm for the minimum spanning tree problem or the SD heuristic for vertex coloring, both of which have proven to be challenging to scale in software [174] and with custom hardware [245]. Beyond classic graph algorithms, new opportunities for fine-grain irregular parallelism may lie in emerging and classic domains: machine learning (e.g., Gibbs sampling in sparse graphical models), numerical optimization (e.g., linear and integer programming approximations), aerial drones (e.g., 3D exploration and navigation), electronic CAD (e.g., place and route), computational geometry (e.g., Voronoi diagrams), bioinformatics, and more.

Compiler-aided program parallelization: While our declarative approach to task ordering makes Swarm programs straightforward to port from sequential implementations, programmers must still invest the time to delimit code into tasks. Fortunately, this thesis offers a new set of capabilities to compiler-aided program parallelization, a decades-old research area that yielded limited results. Relying on static analysis alone, parallelizing compilers have some success finding parallelism in sequential regular algorithms. Even with the dynamic analysis of speculation, speculative parallelizing compilers have struggled to extract much irregular parallelism, as they are curtailed by the limitations of their underlying architectures (i.e., TLS). In contrast, a parallelizing compiler that targets Swarm would benefit from the scalable hardware implementation, including selective aborts, high-throughput ordered commits, and locality-aware execution. The compiler would retain sequential semantics by breaking code into timestamp-ordered tasks, while composing nested loops and function calls with Fractal domains. The resulting code could reduce data movement and isolate contentious accesses with automatically generated spatial hints. This could be a promising approach perform whole-program parallelization to aid programmers in expression of fine-grain irregular parallelism.

Bibliography

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

[2] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-Purpose Graphics Processor Architectures*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[3] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI-12*, 2016.

[4] M. Abadi, M. Isard, and D. G. Murray, "A computational model for TensorFlow: An introduction," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 2017.

[5] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez, "SAM: Optimizing multithreaded cores for speculative parallelism," in *Proc. PACT-26*, 2017.

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

[7] 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.

[8] 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.

[9] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: the end of the road for conventional microarchitectures," in *Proc. ISCA-27*, 2000.

[10] G. A. Agha, "Actors: a model of concurrent computation in distributed systems," Ph.D. dissertation, Massachusetts Institute of Technology, 1985.

[11] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Proc. IPDPS*, 2010.

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

[13] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proc. MICRO-31*, 1998.

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

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the American Federation of Information Processing Societies Spring Joint Computer Conference*, ser. AFIPS, 1967.

[16] *4096x128 ternary CAM datasheet (28nm)*, Analog Bits, 2011.

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

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

[19] 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.

[20] L. B. Arimilli, B. Blaner, B. C. Drerup, C. F. Marino, D. E. Williams, E. N. Lais, F. A. Campisano, G. L. Guthrie, M. S. Floyd, R. B. Leavens, S. M. Willenborg, R. Kalla, and B. Abali, "IBM POWER9 processor and system features for computing in the cognitive era," *IBM Journal of Research and Development*, vol. 62, no. 4/5, 2018.

[21] 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.

[22] Arvind and K. P. Gostelow, "The U-interpreter," *IEEE Computer*, vol. 15, no. 2, 1982.

[23] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, 1990.

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

[25] U. Aydonat and T. S. Abdelrahman, “Hardware support for relaxed concurrency control in transactional memory,” in *Proc. MICRO-43*, 2010.

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

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

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

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

[30] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, “Tiramisu: A polyhedral compiler for expressing fast and portable code,” in *Proc. CGO*, 2019.

[31] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, “Near-data processing: Insights from a MICRO-46 workshop,” *IEEE Micro*, vol. 34, no. 4, 2014.

[32] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, *Multi-Core Cache Hierarchies*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[33] A. Baldassin, E. Borin, and G. Araujo, “Performance implications of dynamic memory allocators on transactional memory systems,” in *Proc. PPoPP*, 2015.

[34] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, “Automatic program parallelization,” *Proceedings of the IEEE*, vol. 81, no. 2, 1993.

[35] P. Barnes Jr, C. Carothers, D. R. Jefferson, and J. LaPre, “Warp speed: executing time warp on 1,966,080 cores,” in *Proc. PADS*, 2013.

[36] L. A. Barroso, U. HÃűlzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3rd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2018.

[37] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: A scalable architecture based on single-chip multiprocessing,” in *Proc. ISCA-27*, 2000.

[38] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proc. SC12*, 2012.

[39] L. Baugh and C. Zilles, "An analysis of I/O and syscalls in critical sections and their implications for transactional memory," in *Proc. ISPASS*, 2008.

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

[41] C. J. Beckmann and C. D. Polychronopoulos, "Fast barrier synchronization hardware," in *Proc. SC90*, 1990.

[42] J. Bennett and S. Lanning, "The Netflix prize," in *KDD Cup and Workshop*, 2007.

[43] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, 1996.

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

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

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

[47] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," in *Proc. SPAA*, 2012.

[48] 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.

[49] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, 1996.

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

[51] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, 1999.

[52] 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.

[53] C. Blundell, E. C. Lewis, and M. M. K. Martin, "Subtleties of transactional memory atomicity semantics," *IEEE Computer Architecture Letters*, vol. 5, no. 2, 2006.

[54] O. A. R. Board, "OpenMP application program interface," 2013.

[55] 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.

[56] 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.

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

[58] M. Bohr, “A 30 year retrospective on Dennard’s MOSFET scaling paper,” *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, 2007.

[59] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proc. PLDI*, 2008.

[60] S. Brand and R. Bidarra, “Multi-core scalable and efficient pathfinding with Parallel Ripple Search,” *Computer Animation and Virtual Worlds*, vol. 23, no. 2, 2012.

[61] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, “Speculative out-of-order event processing with software transaction memory,” in *Proc. of the 2nd intl. conf. on Distributed Event-Based Systems*, 2008.

[62] D. Burger, S. W. Keckler, K. e. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder, “Scaling to the End of Silicon with EDGE Architectures,” *IEEE Computer*, vol. 37, no. 7, 2004.

[63] 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.

[64] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, “Robust architectural support for transactional memory in the Power architecture,” in *Proc. ISCA-40*, 2013.

[65] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun, “Transactional collection classes,” in *Proc. PPoPP*, 2007.

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

[67] C. Caşcaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, “Software transactional memory: Why is it only a research toy?” *Communications of the ACM*, vol. 51, no. 11, 2008.

[68] E. Castillo, L. Alvarez, M. Moreto, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero, “Architectural support for task dependence management with flexible software scheduling,” in *Proc. HPCA-24*, 2018.

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

[70] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proc. HPCA-13*, 2007.

[71] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proc. PPoPP*, 2011.

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

[73] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, 2007.

[74] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS digital design," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 4, 1992.

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

[76] 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.

[77] 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.

[78] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, 2017.

[79] 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.

[80] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. EuroSys*, 2012.

[81] B. V. Cherkassky and A. V. Goldberg, "On implementing the push-relabel method for the maximum flow problem," *Algorithmica*, vol. 19, no. 4, 1997.

[82] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proc. ISCA-25*, 1998.

[83] 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.

[84] 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.

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

[86] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, “The scalable commutativity rule: Designing scalable software for multicore processors,” in *Proc. SOSP-24*, 2013.

[87] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, “Optimistic intra-transaction parallelism on chip multiprocessors,” in *Proc. VLDB*, 2005.

[88] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, “Tolerating dependences between large speculative threads via sub-threads,” in *Proc. ISCA-33*, 2006.

[89] R. P. Colwell, R. P. Nix, J. J. O’Donnell, D. B. Papworth, D. B. Papworth, and P. K. Rodman, “A VLIW architecture for a trace scheduling compiler,” in *Proc. ASPLOS-II*, 1987.

[90] G. Contreras and M. Martonosi, “Characterizing and improving the performance of Intel threading building blocks,” in *Proc. IISWC*, 2008.

[91] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. SoCC-1*, 2010.

[92] 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 Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013.

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

[94] C. Curino, E. Jones, Y. Zhang, and S. Madden, “Schism: a workload-driven approach to database replication and partitioning,” *Proceedings of the VLDB Endowment*, vol. 3, 2010.

[95] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *IEEE Computational Science and Engineering*, vol. 5, no. 1, 1998.

[96] L. Dalessandro and M. L. Scott, “Strong isolation is a weak idea,” in *TRANSACT*, 2009.

[97] L. Dalessandro and M. L. Scott, “Sandboxing transactional memory,” in *Proc. PACT-21*, 2012.

[98] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, “Architecture of a message-driven processor,” in *Proc. ISCA-14*, 1987.

[99] T. David, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask,” in *Proc. SOSP-24*, 2013.

[100] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[101] 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.

[102] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, 2008.

[103] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted MOSFETs with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, 1974.

[104] J. B. Dennis, “First version of a data flow procedure language,” in *Proc. Programming Symposium*, 1974.

[105] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *Proc. ISCA*, 1975.

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

[107] L. Dhulipala, G. Blelloch, and J. Shun, “Julienne: A framework for parallel graph algorithms using work-efficient bucketing,” in *Proc. SPAA*, 2017.

[108] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, “Simplifying concurrent algorithms by exploiting hardware transactional memory,” in *Proc. SPAA*, 2010.

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

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

[111] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, 1959.

[112] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[113] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang, “Software behavior oriented parallelization,” in *Proc. PLDI*, 2007.

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

[115] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui, “Why STM can be more than a research toy,” *Communications of the ACM*, vol. 54, no. 4, 2011.

[116] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, “Preventing versus curing: avoiding conflicts in transactional memories,” in *Proc. PODC*, 2009.

[117] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir, “On the power of hardware transactional memory to simplify memory management,” in *Proc. PODC*, 2011.

[118] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, “Near-threshold computing: Reclaiming Moore’s law through energy efficient integrated circuits,” *Proceedings of the IEEE*, vol. 98, no. 2, 2010.

[119] P. Dudnik and M. M. Swift, “Condition variables and transactional memory: Problem or opportunity?” in *Proc. TRANSACT*, 2009.

[120] 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*, vol. 21, no. 02, 2011.

[121] A. Duran, J. Corbalán, and E. Ayguadé, “Evaluation of OpenMP task scheduling strategies,” in *Proc. of the 4th International Workshop on OpenMP*, 2008.

[122] L. Durant, O. Girouxa, M. Harris, and N. Stam, “Inside Volta: The world’s most advanced data center GPU,” in *NVIDIA Developer Blog*, May 2017.

[123] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Cambridge University Press, 2010.

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

[125] M. Emami, R. Ghiya, and L. J. Hendren, “Context-sensitive interprocedural points-to analysis in the presence of function pointers,” in *Proc. PLDI*, 1994.

[126] N. Enright Jerger, T. Krishna, and L.-S. Peh, *On-Chip Networks*, 2nd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[127] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. ISCA-38*, 2011.

[128] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, “A survey on thread-level speculation techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, 2016.

[129] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, and M. Valero, “Task Superscalar: An Out-of-Order Task Pipeline,” in *Proc. MICRO-43*, 2010.

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

[131] P. Feautrier and C. Lengauer, “Data flow graphs,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011.

[132] P. Feautrier and C. Lengauer, “Polyhedron model,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011.

[133] 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 (TOPLAS)*, vol. 9, no. 3, 1987.

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

[135] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, “Intel® AVX: New frontiers in performance improvements and energy efficiency,” Intel Corporation, White Paper, May 2008.

[136] J. A. Fisher, “Very long instruction word architectures and the ELI-512,” in *Proc. ISCA-10*, 1983.

[137] J. Fix, N. P. Nagendra, S. Apostolakis, H. Zhang, S. Qiu, and D. I. August, “Hardware multithreaded transactions,” in *Proc. ASPLOS-XXIII*, 2017.

[138] M. Franklin and G. S. Sohi, “The expandable split window paradigm for exploiting fine-grain parallelism,” in *Proc. ISCA-19*, 1992.

[139] M. Franklin and G. S. Sohi, “ARB: A hardware mechanism for dynamic reordering of memory references,” *IEEE Transactions on Computers*, vol. 45, no. 5, 1996.

[140] M. Franklin, “The multiscalar architecture,” Ph.D. dissertation, University of Wisconsin–Madison, 1993.

[141] M. Fredman and R. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” in *Proc. of the 25th Symposium on Foundations of Computer Science (FOCS)*, 1984.

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

[143] R. Fujimoto, “The virtual time machine,” in *Proc. SPAA*, 1989.

[144] R. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, vol. 33, no. 10, 1990.

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

[146] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, “Dynamic memory disambiguation using the memory conflict buffer,” in *Proc. ASPLOS-VI*, 1994.

[147] A. García-Yágüez, D. R. Llanos, and A. González-Escribano, “Squashing alternatives for software-based speculative parallelization,” *IEEE Transactions on Computers*, vol. 63, no. 7, 2014.

[148] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *Communications of the ACM*, vol. 53, no. 11, 2010.

[149] S. Garold, “Detection and parallel execution of independent instructions,” *IEEE Transactions on Computers*, vol. 19, no. 10, 1970.

[150] 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.

[151] 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.

[152] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, “XKaapi: A runtime system for data-flow task programming on heterogeneous architectures,” in *Proc. IPDPS*, 2013.

[153] P. P. Gelsinger, “Microprocessors for the new millennium: Challenges, opportunities, and new frontiers,” in *Proc. ISSCC*, 2001.

[154] S. Ghemawat and P. Menage, “TCMalloc: Thread-caching malloc,” https://gperftools.github.io/gperftools/tcmalloc.html, archived at https://perma.cc/EK9E-LBYU, 2007.

[155] 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.

[156] D. Goldfarb and M. D. Grigoriadis, “A computational comparison of the dinic and network simplex methods for maximum flow,” *Annals of Operations Research*, vol. 13, no. 1, 1988.

[157] M. Gonzalez-Mesa, E. Gutierrez, E. L. Zapata, and O. Plata, “Effective Transactional Memory Execution Management for Improved Concurrency,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, 2014.

[158] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, “Speculative versioning cache,” in *Proc. HPCA-4*, 1998.

[159] D. Greenhill, R. Ho, D. Lewis, H. Schmit, K. H. Chan, A. Tong, S. Atsatt, D. How, P. McElheny, K. Duwel, J. Schulz, D. Faulkner, G. Iyer, G. Chen, H. K. Phoon, H. W. Lim, W. Koay, and T. Garibay, “A 14nm 1GHz FPGA with 2.5D transceiver integration,” in *Proc. ISSCC*, 2017.

[160] T. Grosser, A. Größlinger, and C. Lengauer, “Polly - performing polyhedral optimizations on a low-level intermediate representation,” *Parallel Processing Letters*, vol. 22, no. 4, 2012.

[161] 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.

[162] R. Guerraoui and M. Kapałka, *Principles of Transactional Memory*, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[163] R. Guerraoui and M. Kapałłka, “On the correctness of transactional memory,” in *Proc. PPoPP*, 2008.

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

[165] G. Gupta and G. S. Sohi, “Dataflow execution of sequential imperative programs on multicore architectures,” in *Proc. MICRO-44*, 2011.

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

[167] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, “Programming with transactional coherence and consistency (TCC),” in *Proc. ASPLOS-XI*, 2004.

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

[169] 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.

[170] F. M. Harper and J. A. Konstan, “The MovieLens datasets: History and context,” *ACM Transactions on Interactive Intelligent Systems (TiiS)*, vol. 5, no. 4, 2015.

[171] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed., ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[172] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, “Composable memory transactions,” in *Proc. PPoPP*, 2005.

[173] P. Hart, N. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968.

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

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

[176] 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.

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

[178] L. J. Hendren and A. Nicolau, “Parallelizing programs with recursive data structures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, 1990.

[179] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Elsevier, 2019.

[180] M. Herlihy, “Wait-free synchronization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, 1991.

[181] M. Herlihy, “Transactional memories,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011.

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

[183] M. Herlihy and Y. Sun, “Distributed transactional memory for metric-space networks,” *Distributed Computing*, vol. 20, no. 3, 2007.

[184] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, “Evaluating collaborative filtering recommender systems,” *ACM Transactions on Information Systems (TOIS)*, vol. 22, no. 1, 2004.

[185] M. D. Hill and M. Marty, “Amdahl’s Law in the Multicore Era,” *IEEE Computer*, vol. 41, no. 7, 2008.

[186] W. D. Hillis and G. L. Steele, Jr., “Data parallel algorithms,” *Communications of the ACM*, vol. 29, no. 12, 1986.

[187] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?” in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

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

[189] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein, “Scaling, power, and the future of CMOS,” in *Proc. IEDM*, 2005.

[190] S. Horwitz, P. Pfeiffer, and T. Reps, “Dependence analysis for pointer variables,” in *Proc. PLDI*, 1989.

[191] D. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas, “Two hardware-based approaches for deterministic multiprocessor replay,” *Communications of the ACM*, 2009.

[192] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, “Decoupled software pipelining creates parallelization opportunities,” in *Proc. CGO*, 2010.

[193] J. Hubicka, A. Jaeger, and M. Mitchell, “System V application binary interface,” *AMD64 Architecture Processor Supplement*, 2013.

[194] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg, “McRT-Malloc: A scalable transactional memory allocator,” in *Proc. ISMM*, 2006.

[195] N. Hunt, P. S. Sandhu, and L. Ceze, “Characterizing the performance and energy efficiency of lock-free data structures,” in *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*, 2011.

[196] Intel, “TBB http://www.threadingbuildingblocks.org.”

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

[198] C. Jacobi, T. Slegel, and D. Greiner, “Transactional memory architecture and implementation for ibm system z,” in *Proc. MICRO-45*, 2012.

[199] 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.

[200] D. R. Jefferson, “Virtual time,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, 1985.

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

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

[203] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, “Unlocking ordered parallelism with the Swarm architecture,” *IEEE Micro*, vol. 36, no. 3, 2016.

[204] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez, “Harmonizing speculative and non-speculative execution in architectures for ordered parallelism,” in *Proc. MICRO-51*, 2018.

[205] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, “Advances in dataflow programming languages,” *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, 2004.

[206] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” in *Proc. ISCA-44*, 2017.

[207] 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*, vol. 50, no. 2, 1999.

[208] A. Kägi, D. Burger, and J. R. Goodman, “Efficient synchronization: Let them eat QOLB,” in *Proc. ISCA-24*, 1997.

[209] A. Kannan, N. E. Jerger, and G. H. Loh, “Enabling interposer-based disintegration of multi-core processors,” in *Proc. MICRO-48*, 2015.

[210] 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.

[211] 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.

[212] T. Knight, "An architecture for mostly functional languages," in *Proc. of the ACM Conference on LISP and Functional Programming*, 1986.

[213] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proc. PLDI*, 2018.

[214] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Transactions on Computers*, vol. 48, no. 9, 1999.

[215] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, 1985.

[216] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, 1956.

[217] 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.

[218] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali, "Exploiting the commutativity lattice," in *Proc. PLDI*, 2011.

[219] 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.

[220] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proc. PLDI*, 2007.

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

[222] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, no. 1, 1982.

[223] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, 1981.

[224] B. C. Kuszmaul, "SuperMalloc: A super fast multithreaded malloc for 64-bit machines," in *Proc. ISMM*, 2015.

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

[226] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, no. 5, 2006.

[227] E. A. Lee and D. G. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, 1987.

[228] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Transactions on Computers*, vol. 36, no. 1, 1987.

[229] V. Leis, A. Kemper, and T. Neumann, “Exploiting hardware transactional memory in main-memory databases,” in *Proc. of IEEE 30th International Conference on Data Engineering*, 2014.

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

[231] A. Lenharth, D. Nguyen, and K. Pingali, “Priority queues are not good concurrent priority schedulers,” in *Proc. of the European Conference on Parallel Processing (Euro-Par)*, 2015.

[232] K. Lepak, G. Talbot, S. White, N. Beck, and S. Naffziger, “The next generation AMD enterprise server product architecture,” in *Proc. HotChips*, 2017.

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

[234] A. Lew, J. Marsden, M. Ortiz, and M. West, “Asynchronous variational integrators,” *Arch. Rational Mech. Anal.*, vol. 167, no. 2, 2003.

[235] C. Lin and L. Snyder, “ZPL: An array sublanguage,” in *Proceedings of the 6th international workshop on Languages and Compilers for Parallel Computing*, 1993.

[236] M. H. Lipasti and J. P. Shen, “Exceeding the dataflow limit via value prediction,” in *Proc. MICRO-29*, 1996.

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

[238] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, “POSH: a TLS compiler that exploits program structure,” in *Proc. PPoPP*, 2006.

[239] D. B. Lomet, “Process structuring, synchronization, and recovery using atomic actions,” in *Proc. of the ACM Conference on Language Design for Reliable Software*, 1977.

[240] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in *Proc. ASPLOS-XIII*, 2008.

[241] B. Lucia, J. Devietti, T. Bergan, L. Ceze, and D. Grossman, “Lock prediction,” in *2nd USENIX Workshop on Hot Topics in Parallelism*, 2010.

[242] 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.

[243] D. Makreshanski, J. Levandoski, and R. Stutsman, "To lock, swap, or elide: On the interplay of hardware transactional memory and lock-free indexing," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, 2015.

[244] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *Proc. ICS'98*, 1998.

[245] A. Mariano, D. Lee, A. Gerstlauer, and D. Chiou, "Hardware and software implementations of Prim's algorithm for efficient minimum spanning tree computation," in *Proc. of the International Embedded Systems Symposium*, 2013.

[246] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," in *Proc. SIGGRAPH*, 2003.

[247] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, 2012.

[248] J. F. Martínez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," in *Proc. ASPLOS-X*, 2002.

[249] T. Maruyama, Y. Akizuki, T. Tabata, K. Kitamura, N. Takagi, H. Ishii, S. Watanabe, and F. Tawa, "SPARC64 XII: Fujitsu's latest 12-core processor for mission-critical servers," *IEEE Micro*, vol. 38, no. 5, 2018.

[250] 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)*, vol. 48, no. 2, 2015.

[251] 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.

[252] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali, "Structure-driven optimizations for amorphous data-parallel programs," in *Proc. PPoPP*, 2010.

[253] U. Meyer and P. Sanders, "Delta-stepping: A parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, 2003.

[254] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proc. SPAA*, 2002.

[255] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proc. PODC*, 1996.

[256] S. P. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2012.

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

[258] 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.

[259] J. Misra, "Distributed discrete-event simulation," *ACM Computing Surveys (CSUR)*, vol. 18, no. 1, 1986.

[260] M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications*, D. P. Mehta and S. Sahni, Eds. Chapman & Hall/CRC, 2004.

[261] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, 1965.

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

[263] L. Morais, V. Silva, A. Goldman, C. Alvarez, J. Bosch, M. Frank, and G. Araujo, "Adding tightly-integrated task scheduling acceleration to a RISC-V multi-core processor," in *Proc. MICRO-52*, 2019.

[264] 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.

[265] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *Proc. ISCA*, 1997.

[266] J. E. B. Moss, "Open nested transactions: Semantics and support," in *Workshop on Memory Performance Issues*, 2006.

[267] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and architecture sketches," *Science of Computer Programming*, vol. 63, no. 2, 2006.

[268] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase Reconciliation for Contended In-Memory Transactions." in *Proc. OSDI-11*, 2014.

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

[270] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. SOSP-24*, 2013.

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

[272] 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.

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

[274] 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.

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

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

[277] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-shelf hardware transactional memory," in *Proc. IISWC*, 2014.

[278] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proc. ASPLOS-VII*, 1996.

[279] K. Olukotun and L. Hammond, "The future of microprocessors," *ACM Queue*, vol. 3, no. 7, 2005.

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

[281] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. MICRO-38*, 2005.

[282] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, "Controlling program execution through binary instrumentation," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, 2005.

[283] R. Panigrahy and S. Sharma, "Sorting and searching using ternary CAMs," *IEEE Micro*, vol. 23, no. 1, 2003.

[284] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proc. SPAA*, 2011.

[285] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, 1979.

[286] Y. N. Patt, W. M. Hwu, and M. Shebanow, "HPS, a new microarchitecture: Rationale and introduction," in *Proc. MICRO-18*, 1985.

[287] D. Patterson, "50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set," in *Proc. ISSCC*, 2018.

[288] J. M. Perez, R. M. Baida, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *IEEE International Conference on Cluster Computing*, 2008.

[289] 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.

[290] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, 1987.

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

[292] M. Postiff, D. Greene, G. Tyson, and T. Mudge, "The limits of instruction level parallelism in SPEC95 applications," *Comp. Arch. News*, vol. 27, no. 1, 1999.

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

[294] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas, "Removing architectural bottlenecks to the scalability of speculative parallelization," in *Proc. ISCA-28*, 2001.

[295] X. Qian, W. Ahn, and J. Torrellas, "ScalableBulk: Scalable cache coherence for atomic blocks in a lazy environment," in *Proc. MICRO-43*, 2010.

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

[297] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," in *Proc. PLDI*, 2005.

[298] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. PLDI*, 2013.

[299] 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.

[300] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proc. ASPLOS-X*, 2002.

[301] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proc. ISCA-32*, 2005.

[302] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Proc. MICRO-41*, 2008.

[303] 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.

[304] N. Rapolu, K. Kambatla, S. Jagannathan, and A. Grama, "TransMR: Data-centric programming beyond data parallelism," in *HotCloud*, 2011.

[305] B. R. Rau and J. A. Fisher, "Instruction-level parallel processing: History, overview, and perspective," *The Journal of Supercomputing*, vol. 7, no. 1, 1993.

[306] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proc. PLDI*, 1995.

[307] 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.

[308] 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.

[309] 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.

[310] R. Riedlinger, R. Arnold, L. Biro, B. Bowhill, J. Crop, K. Duda, E. S. Fetzer, O. Franza, T. Grutkowski, C. Little, C. Morganti, G. Moyer, A. Munch, M. Nagarajan, C. Parks, C. Poirier, B. Repasky, E. Roytman, T. Singh, and M. W. Stefaniw, "A 32 nm, 3.1 billion transistor, 12 wide issue Itanium® processor for mission-critical servers," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, 2012.

[311] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: MultiQueues: Simple relaxed concurrent priority queues," in *Proc. SPAA*, 2015.

[312] M. C. Rinard, D. J. Scales, and M. S. Lam, "Jade: a high-level, machine-independent language for parallel programming," *IEEE Computer*, vol. 26, no. 6, 1993.

[313] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *ACM TOPLAS*, vol. 19, no. 6, 1997.

[314] Y. Robert, “Task graph scheduling,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011.

[315] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel, “TxLinux: Using and managing hardware transactional memory in an operating system,” in *Proc. SOSP-21*, 2007.

[316] C. J. Rossbach, O. S. Hofmann, and E. Witchel, “Is transactional programming actually easier?” in *Proc. PPoPP*, 2010.

[317] R. M. Russell, “The CRAY-1 computer system,” *Communications of the ACM*, vol. 21, no. 1, 1978.

[318] M. M. Saad and B. Ravindran, “Hyflow: A high performance distributed software transactional memory framework,” in *Proceedings of the 20th international symposium on High performance distributed computing*, 2011.

[319] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, “IBM Power9 processor architecture,” *IEEE Micro*, vol. 37, no. 2, 2017.

[320] D. Sainz and H. Attiya, “Relstm: A proactive transactional memory scheduler,” in *International Workshop on Transactional Computing (TRANSACT)*, 2013.

[321] J. H. Salz, R. Mirchandaney, and K. Crowley, “Run-time parallelization and scheduling of loops,” *IEEE Transactions on Computers*, vol. 40, no. 5, 1991.

[322] Samsung, “Samsung SSD 960 PRO M.2 Data Sheet,” 2017.

[323] D. Sanchez and C. Kozyrakis, “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proc. ISCA-40*, 2013.

[324] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, “Dynamic Fine-Grain Scheduling of Pipeline Parallelism,” in *Proc. of the 20th intl conf. on Parallel Architectures and Compilation Techniques*, 2011.

[325] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, “Implementing signatures for transactional memory,” in *Proc. MICRO-40*, 2007.

[326] D. Sanchez, R. Yoo, and C. Kozyrakis, “Flexible architectural support for fine-grain scheduling,” in *Proc. ASPLOS-XV*, 2010.

[327] T. Sato, K. Ohno, and H. Nakashima, “A mechanism for speculative memory accesses following synchronizing operations,” in *Proc. IPDPS*, 2000.

[328] M. Schlansker, T. M. Conte, J. Dehnert, K. Ebcioglu, J. Z. Fang, and C. L. Thompson, “Compilers for instruction-level parallelism,” *IEEE Computer*, vol. 30, no. 12, 1997.

[329] M. L. Scott, *Shared-Memory Synchronization*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[330] S. L. Scott, "Synchronization and Communication in the T3E Multiprocessor," in *Proc. ASPLOS-VII*, 1996.

[331] M. Shafique and S. Garg, "Computing in the dark silicon era: Current trends and research challenges," *IEEE Design & Test*, vol. 34, no. 2, 2017.

[332] H. Sharangpani and K. Arora, "Itanium processor microarchitecture," *IEEE Micro*, vol. 20, no. 5, 2000.

[333] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. PODC*, 1995.

[334] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, 2016.

[335] J. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, C. Hwang, H. Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A. Leon, and A. Strong, "A 40nm 16-core 128-thread CMT SPARC SoC processor," in *Proc. ISSCC*, 2010.

[336] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proc. ISCA-35*, 2008.

[337] J. Shun, *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. ACM and Morgan & Claypool Publishers, 2017.

[338] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.

[339] 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.

[340] 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.

[341] T. Skare and C. Kozyrakis, "Early release: Friend or foe?" in *Proc. WTW*, 2006.

[342] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, no. 12, 1995.

[343] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights Landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, 2016.

[344] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. ISCA-22*, 1995.

[345] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[346] L. Soule and A. Gupta, "An evaluation of the Chandy-Misra-Bryant algorithm for digital logic simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 1, no. 4, 1991.

[347] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 3, 2005.

[348] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proc. ISCA-27*, 2000.

[349] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proc. HPCA-4*, 1998.

[350] J. S. Steinman, "Breathing time warp," in *Proc. PADS*, 1993.

[351] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the Oklahoma update," *IEEE Parallel Distributed Technology: Systems Applications*, vol. 1, no. 4, 1993.

[352] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *Proc. ISCA-44*, 2017.

[353] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 1, 2009.

[354] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," in *Proc. IPDPS*, 2003.

[355] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "A lock-free algorithm for concurrent bags," in *Proc. SPAA*, 2011.

[356] G. J. Sussman and G. L. Steele Jr, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, 1998.

[357] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[358] H. Sutter, "Lock-free code: A false sense of security," *Dr. Dobb's Journal*, vol. 33, no. 9, September 2008.

[359] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proc. MICRO-36*, 2003.

[360] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, 2007.

[361] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang, "SkyLake-SP: A 14nm 28-core Xeon processor," in *Proc. ISSCC*, 2018.

[362] A. S. Tanenbaum and D. J. Wetherall, *Computer networks*, 5th ed., P. Hall, Ed., 2010.

[363] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP tasks in Nanos V4," in *Proc. of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2007.

[364] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proc. of the 10th International Conference on Compiler Construction*, 2002.

[365] M. Thorup, "On RAM priority queues," in *Proc. SODA*, 1996.

[366] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[367] C. Tian, C. Lin, M. Feng, and R. Gupta, "Enhanced speculative parallelization via incremental recovery," in *Proc. PPoPP*, 2011.

[368] J. Torrellas, "Thread-level speculation," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011.

[369] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew, "The superthreaded processor architecture," *IEEE Transactions on Computers*, vol. 48, no. 9, 1999.

[370] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Compiler techniques for the superthreaded architectures," *International Journal of Parallel Programming*, vol. 27, no. 1, 1999.

[371] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. SOSP-24*, 2013.

[372] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. ISCA-22*, 1995.

[373] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly," in *Proc. ASPLOS-XXIII*, 2017.

[374] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, 1993.

[375] N. Vachharajani, "Intelligent speculation for pipelined multithreading," Ph.D. dissertation, Princeton University, 2008.

[376] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[377] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, 1990.

[378] A. Varga and R. Hornig, "An overview of the OMNeT++ simulation environment," in *Proc. of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ser. Simutools, 2008.

[379] A. Varga and A. Şekercioğlu, "Parallel simulation made easy with OMNeT++," in *Proc. of the European Simulation Multiconference*, ser. ESM, 2001.

[380] A. H. Veen, "Dataflow machine architecture," *ACM Computing Surveys (CSUR)*, vol. 18, no. 4, 1986.

[381] T. N. Vijaykumar and G. S. Sohi, "Task selection for a Multiscalar processor," in *Proc. MICRO-31*, 1998.

[382] H. Volos, N. Goyal, and M. M. Swift, "Pathological interaction of locks with transactional memory," in *TRANSACT*, 2008.

[383] 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.

[384] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. ISCA-19*, 1992.

[385] J. von Neumann, "First draft of a report on the EDVAC," 1945, reprinted in [**?**].

[386] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, 1997.

[387] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, no. 7589, 2016.

[388] D. Wall, "Limits of instruction-level parallelism," in *Proc. ASPLOS-IV*, 1991.

[389] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proc. PACT-21*, 2012.

[390] C. Wang, Y. Liu, and M. Spear, "Transaction-friendly condition variables," in *Proc. SPAA*, 2014.

[391] 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*, vol. 10, no. 1, 1967.

[392] 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*, vol. 27, no. 5, 2007.

[393] M. Wimmer, F. Versaci, J. Träff, D. Cederman, and P. Tsigas, “Data structures for task-based priority scheduling,” in *Proc. PPoPP*, 2014.

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

[395] M. Xu, R. Bodik, and M. D. Hill, “A flight data recorder for enabling full-system multiprocessor deterministic replay,” in *Proc. ISCA-30*, 2003.

[396] C. Yang and B. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in *Proc. ICDCS*, 1988.

[397] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, “Hybrid Dataflow/Von-Neumann Architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, 2014.

[398] 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 *Proceedings of the 2007 international workshop on Parallel symbolic computation*, 2007.

[399] 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.

[400] 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.

[401] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, “Performance evaluation of Intel® transactional synchronization extensions for high-performance computing,” in *Proc. SC13*, 2013.

[402] R. M. Yoo and H.-H. S. Lee, “Adaptive transaction scheduling for transactional memory systems,” in *Proc. SPAA*, 2008.

[403] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, “Staring into the abyss: An evaluation of concurrency control with one thousand cores,” *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 2014.

[404] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *Proc. ASPLOS-X*, 2002.

[405] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of memory-resident value communication between speculative threads," in *Proc. CGO*, 2004.

[406] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors," in *Proc. HPCA-5*, 1999.

[407] Y. Zhang, V. Kiriansky, C. Mendis, S. P. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE BigData*, 2017.

[408] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "GraphIt: A high-performance graph DSL," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[409] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar, "Reducing task creation and termination overhead in explicitly parallel programs," in *Proc. PACT-19*, 2010.

[410] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proc. ISCA-34*, 2007.

[411] C. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," in *TRANSACT*, 2006.