

Compiler-Hardware Co-Design for Pervasive Parallelization

by

Victor A. Ying

B.S.E., Princeton University (2016)

S.M., Massachusetts Institute of Technology (2019)

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

Doctor of Philosophy

at the

Massachusetts Institute of Technology

September 2023

© 2023 Victor A. Ying. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable,
royalty-free license to exercise any and all rights under copyright,
including to reproduce, preserve, distribute, and publicly display
copies of the thesis, or release the thesis under an open-access license.

Authored by:	Victor A. Ying Department of Electrical Engineering and Computer Science June 30, 2023
Certified by:	Daniel Sanchez Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejwski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Compiler-Hardware Co-Design for Pervasive Parallelization

by

Victor A. Ying

Submitted to the Department of Electrical Engineering and Computer Science
on June 30, 2023, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

Modern computer systems have hundreds of processor cores, so highly parallel programs are critical for achieving high performance. However, writing parallel programming remains difficult on current systems, so many programs are still sequential. This dissertation presents new compilers and hardware architectures that can parallelize complex programs while retaining the simplicity of sequential code. Our new systems allow real-world programs to use hundreds of cores without burdening programmers with deadlock or data races.

This dissertation follows a novel approach that eliminates the burden of explicit parallel programming to make parallel execution pervasive. This approach relies on four guiding principles. First, exploiting *implicit* parallelism preserves the simplicity of sequential execution. Second, dividing computation into *tiny tasks*, as short as tens of instructions each, unlocks plentiful fine-grained parallelism in challenging programs. Hardware-compiler co-design techniques can create many tasks in parallel and reduce per-task overheads to make tiny tasks scale to many cores. Third, new hardware and software mechanisms can *compose* parallelism across entire programs, removing serializing barriers to overlap executions of nested parallel subroutines. Finally, exploiting static and dynamic information for *data locality* reduces data movement costs while maintaining load balance on large multicore systems.

This dissertation presents three systems that embody these four principles. First, T4 introduces automatic program transformations that exploit a novel hardware architecture to parallelize sequential programs. As a result, T4 scales hard-to-parallelize real-world programs to tens of cores, resulting in order-of-magnitude speedups. Second, S5 builds on T4 with novel transformations to remove needless serialization in a broad class of challenging data structures. Thus, S5 scales complex real-world programs to hundreds of cores, delivers additional order-of-magnitude speedups over T4, and outperforms manually parallelized code tuned by experts. Finally, ASH is an accelerator that demonstrates the same approach can be applied with simpler mechanisms tailored for digital circuit simulation. A small ASH implementation is $32\times$ faster than a large multicore CPU running a state-of-the-art parallel simulator.

Thesis Supervisor:

Daniel Sanchez

Professor of Electrical Engineering and Computer Science

Acknowledgments

This work would be impossible without the collaborative efforts of many individuals who have contributed to the Swarm project over the last decade, as well as a wonderful community who gave me the opportunity to spend years chipping away at challenging research goals.

I have had fantastic mentorship at MIT. My supervisor, Prof. Daniel Sanchez, is a great source of insight, an impressive software engineer, and a walking library of computer architecture research. He was a reliable source of encouragement when I wanted to invest long months in learning to build compilers before I was able to produce useful tools or interesting performance improvements. Daniel is also great at demanding and showing the way to produce high quality work and thoughtful writing that shows the value of research.

I was also lucky enough to collaborate multiple times with Prof. Joel Emer and spend one summer gaining experience at an industry research lab in Joel’s group at Nvidia. Joel made a great impression on me when I was a new graduate student, with his care for teaching and for distilling general concepts out of complex engineering projects to enable us to see new research opportunities.

Prof. Saman Amarasinghe runs a great compiler research group and it was a joy to collaborate with him and his students on the GraphIt team, Ajay Brahmakshatriya, Claire Hsu, and Yunming Zhang, to combine our technologies into a unified compiler framework. All three members of my Ph.D. committee, Daniel, Joel, and Saman, provided invaluable feedback not only on this dissertation but also on the preceding years of work. My academic advisor, Prof. Charles Leiserson, has also offered wisdom that has brought me great benefits in my academic path and career planning.

My work has directly built on that of many peers and collaborators. My predecessors Mark Jeffrey, Suvinay Subramanian, and Maleen Abeydeera created the Swarm architecture that this dissertation builds on, and they were crucial mentors for the first half of my time at MIT. I am thankful for all I have learned from long hours spent brainstorming, designing, writing, and reflecting on the state of computer architecture. Many others students in the Sanchez group have contributed to Swarm as well. In particular, my collaborations with Ryan Lee, Fares Elsabbagh, Shabnam Sheikhha, Quan Nguyen, Domenic Natile, and Robert Durfee have shaped the work described in this dissertation. In

particular, Fares, Shabnam, and Quan each made large contributions to the development of ASH. Others at MIT such as T.B. Schardl played a significant role in teaching me of the feasibility of developing compilers that aggressively optimize fine-grained parallel programs.

I am also indebted to the rest of the Sanchez lab, including my fellow Ph.D. students Po-An Tsai, Guowei Zhang, Anurag Mukkara, Axel Feldmann, Yifan Yang, Nikola Samardzic, and Maggie Du, as well as undergraduates and M.Eng. students Aleksandar Krastev, Nithya Attaluri, Elliott Forde, Grace Yin, Matthew Ha, Jordan Docter, and Michael Gilbert. They made it worth it to go into the office every day. I would never have completed these projects without the presence of these intelligent and kind individuals. I am also thankful to the National Science Foundation, Sony, and DARPA for funding my research projects.

I have also had so much support from family, friends, and teachers in going to graduate school. My undergraduate thesis advisor, Prof. Sharad Malik, provided a critical push for me to find a research field I liked, and was the first to introduce me to then-Assistant Professor Sanchez. My off-campus friends Nick Rance, Isobel Green, and Chad Wangsanuwat have been critical in supporting me in good times and bad, providing crucial encouragement and company during late nights and weekends. Drs. Austin Egan, Anthony Van Niel, and Yiling Zhang also provided excellent professional help. Last but not least, my family shaped me into the person I am today. My wife Jenny Sun was always there when I needed her and would never let me give up on finishing this Ph.D. She, along with my parents Charles Ying and Jane Zhu and my brother Kevin Ying, made all of it possible.

Contents

Abstract	2
1 Introduction	7
1.1 Challenges	9
1.2 Contributions	12
2 Background and Related Work	15
2.1 General Approaches to Parallelization	15
2.2 Parallelizing Compilers	18
2.3 Swarm	22
3 Trees of Tiny Timestamped Tasks	31
3.1 Introduction	31
3.2 Motivation	33
3.3 T4 Implementation Overview	37
3.4 Eliminating the Call Stack	39
3.5 Full-Program Fine-Grain Parallelization	41
3.6 Parallel Loop Expansion	45
3.7 Reducing Communication Costs	48
3.8 Scalable Speculative Memory Allocation	50
3.9 Putting It All Together	51
3.10 Evaluation	53
3.11 Additional Related Work	63
3.12 Summary	64
4 Schedule Structures to Sidestep Spurious Serialization	66
4.1 Introduction	66
4.2 S5 Makes Implicit Parallelization Effective	72
4.3 S5 Implementation and Optimizations	76
4.4 Fully Automatic Parallelization of Graph Algorithms	88

4.5	Evaluation	92
4.6	Additional Related Work	103
4.7	Summary	107
5	Accelerators for Simulating Hardware	112
5.1	Introduction	112
5.2	Motivation and Background	116
5.3	System Overview	122
5.4	Prioritized Dataflow Execution with DASH	123
5.5	Selective Dataflow Execution	132
5.6	Task-Driven Instruction Prefetching	135
5.7	Hardware Costs	136
5.8	Methodology	136
5.9	Evaluation	139
5.10	Additional Related Work	144
5.11	Summary	148
6	Conclusion and Future Work	149
6.1	Future Work	150
	Bibliography	155

1 Introduction

Fast computation is fundamental to scientific and technological progress, and parallelism is increasingly fundamental to fast computation. Laptops, servers, supercomputers, and smartphones all feature large multicore CPUs, each with tens of billions of transistors, and as many as hundreds of cores. The only way to fully use these large hardware resources is to parallelize computation by dividing it into *tasks* that run simultaneously on different cores. Parallel execution makes computation faster and enables computation to solve larger problems [81]. Making parallelism pervasive is critical to unlocking the growing capability of computer systems to improve our society’s knowledge and quality of life.

After decades of research and real-world practice, parallel programming is used sporadically. Successfully parallelizing a program to use the tens or hundreds of cores available in a modern machine remains rare. Most programmers avoid parallel programming, finding it not effective enough to be worth the effort. Software performance experts parallelize small functions or subroutines only if they are noticed to take significant computation time and only if they can eliminate order constraints that prevent parallelization on current machines [87, 180]. This approach can deliver large benefits to an application whose execution time is dominated by a few parallelizable subroutines. However, in many applications, execution time is distributed widely across a diverse range of code. If merely 2% of execution time is unparallelizable, Amdahl’s law [9, 92] says we can never obtain parallel speedup greater than $\frac{1}{2\%} = 50\times$, dooming any attempt to make use of hundreds of cores. Making most applications highly parallel requires new techniques that are general, easy to use (requiring no or minimal programmer effort), and reliably effective, so they can be applied across the entire program.

This dissertation argues that such pervasive parallelization requires new interfaces between hardware and software. Current general-purpose

processors take an approach that leaves little room for software tools to aid programmers in unlocking parallelism. Today’s predominant shared-memory systems require programmers to use explicit synchronization primitives (e.g., locks and barriers) to perform coordination of parallel tasks. Using these primitives requires careful thought about high-level intentions, and even correctly synchronized subroutines often cannot be safely composed together into a new program [86]. As a result, there is no practical tool that automatically selects the correct synchronization in general-purpose programming languages. Lower-level synchronization primitives (e.g., relaxed atomics [7, 20, 51, 129, 241]) that improve parallelism by conveying precisely when tasks need to be ordered are even more error-prone to use, so manually parallelized programs use them rarely, choosing simpler synchronization that limits parallelism more than is needed. We advocate for hardware to provide a radically different interface to software to make it possible to automatically maximize parallelism in general programs.

With the right interfaces, we claim compilers can arrange tasks and data so hardware can tell precisely when synchronization is or isn’t needed, making parallelization automatic while also unlocking much more performance than is possible on today’s hardware. Our approach relies on exploiting the dynamic information that hardware can discover at runtime. Modern hardware already efficiently tracks and exploits dynamic information, such as in caches with coherence protocols that track where memory is shared to ensure delivery of updated data where needed. This dissertation presents inexpensive extensions to these existing hardware mechanisms that enable a co-designed compiler and hardware system to speculatively execute or synchronize parallel tasks exactly when needed. By reallocating responsibilities so that software and hardware each leverage more of their strengths, parallelization becomes easier and more effective.

To demonstrate the power of our approach, this dissertation presents systems that perform effective parallelization of a wide variety of applications that prior work has found difficult to parallelize. Our systems avoid the complexity of parallel programming with abstractions that maintain the ease and generality of sequential execution. We also leverage composability to maximize the parallelism the system extracts from diverse combinations of tiny tasks, without restricting each task to performing only limited actions.

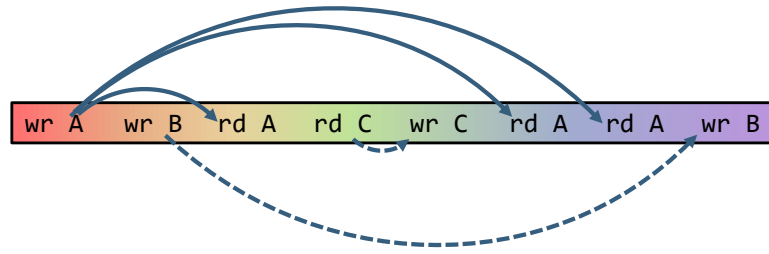
1.1 Challenges

The key challenge in parallelization is executing tasks in a way that respects their *dependences*. There are two types of dependences. First, two tasks have a *control dependence* if the first task can determine whether or not to execute the second [8]. Second, a *data dependence* occurs between any two tasks that access the same state element or memory location, if at least one access is a write [26]. If there is no control or data dependence between them, two tasks are *independent* and can safely run in parallel or out of order.

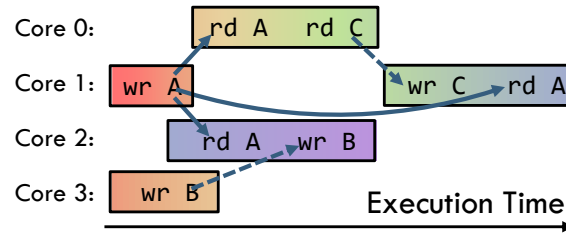
Figure 1-1 shows how a parallelized computation can use multiple cores while ensuring execution order respects data dependences. Accesses to variable A are *true data dependences* because the reads of this variable need to obtain the value from the preceding write. However, the write-after-read and write-after-write dependences that occur on variables B and C, are *false dependences*. False dependences occur because a value is being overwritten, and may be removed by allocating additional memory locations for new values. (As we will see, this reallocation can occur through a variety of techniques such as point-to-point messaging and privatization.) Figure 1-1(c) demonstrates performance could also be improved if the system supports dividing computation into shorter tasks, as each smaller task accesses fewer variables and therefore has fewer dependences restricting when it can run.

Current systems work well only on a restrictive class of applications with easy parallelism, where tasks are large and dependences are rare. No prior system can automatically parallelize real-world workloads with frequent unpredictable dependences and make performance scale up to hundreds of cores. This problem has seen decades of work but several limiting factors cause current systems to fall short:

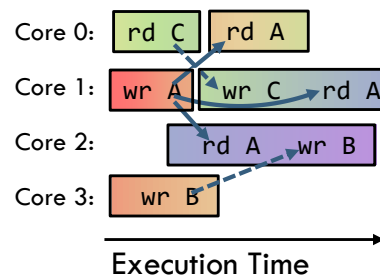
Dependence on brittle static analyses. One common approach to parallelization is to have compilers prove the absence of dependences before tasks can run in parallel [154]. When the compiler cannot prove their absence, dependences are conservatively assumed to be present. Not only does static analysis fail to capture input-dependent behavior, just determining whether a possible execution exists where two tasks will be dependent is undecidable [130, 189]. In practice, compilers perform best-effort analyses [94] and fall back to serial execution under conditions that are difficult for a user to predict without deep knowledge of compiler



(a) An example sequential program with eight memory accesses.



(b) A parallelization of the sequential program with five tasks.



(c) Splitting Core 0's task from (b) in two and allowing the shorter tasks to run out of order improves performance.

Figure 1-1: Parallelization must respect data dependences (arrows, with true data dependences as solid arrows). Parallel performance can improve given flexibility to choose small tasks and execute tasks out of order.

internals. These static parallelization efforts do not generalize to complex real-world code.

Limited hardware support for fine-grained parallelism. Today's multicore CPUs require software to perform synchronization and communication through shared memory [78, 150], which involves waiting hundreds of clock cycles to move cache lines between cores. This can negate the benefits of spawning short tasks [204]. As a result, programmers forgo fine-grained parallelization opportunities. Prior proposals

of hardware support for detecting dependences when parallelizing sequential code used mechanisms that perform task creation or commit one task at a time. These mechanisms limit throughput, especially with tiny tasks [139]. This throughput limitation becomes intolerable as multicores grow from few cores to many because keeping many cores busy requires creating tasks rapidly.

Lack of hardware support for parallel composition. The rigid scheduling of tasks to cores required by most prior hardware proposals for automatic parallelization also limits the forms of parallelism that may be exploited. For example, rather than performing whole-program parallelization, many systems executed no more than one parallelized loop at a time. As a result, there would not be enough tasks to fill a large machine for those executions of the parallelized loop that had too few iterations. Even if the parallelized loop has many iterations, the serial execution occurring between the termination of one parallel loop and the start of the next one prevents the program as a whole from scaling to many cores.

Spurious serialization. In many cases, data dependences are unneeded or false dependences: they occur because memory space is being reused to store a new value rather than because a task needs to consume another task’s previously produced value. These dependences cause serialization of tasks that are performing semantically independent work. There are many causes of such false dependences, from memory layout choices made by a compiler or library implementation to the programmer’s explicit choice of data structures. In principle, false dependences can be eliminated by allocating extra space for new values, but there has not been a systematic approach to optimize low-level data layout or high-level data structures that addresses the need to co-optimize parallelism and locality for programmers who still prefer the ease and generality of writing sequential code.

Locality-insensitive parallelization. As computation scales to tackle large problem sizes using large systems, data movement becomes the dominant factor that limits both performance and energy efficiency. But prior work takes limited information on data locality into account when making decisions about where to schedule the execution of a task. As a result, the extra data movement caused by poor choices of where to schedule tasks causes data-movement overheads to grow with the number of cores used, preventing effective scaling to large system sizes.

Restriction to specific hardware and programming models.

Prior attempts at general automatic parallelization targeted restricted systems, such as considering only cache-coherent general-purpose CPUs. However, two trends lead to increasing diversity in hardware and software. First, hardware specialization is increasingly common. General-purpose systems have overheads that can be eliminated by hardware accelerators designed only to execute particular workloads. These hardware accelerators typically forgo the complexity of cache coherence and require compilers to handle communication explicitly. Second, domain-specific programming frameworks make complex parallel performance optimizations simpler for programmers, in fact increasing performance through library or compiler implementations that exploit awareness of application semantics. These developments decrease the relevance of techniques that are not demonstrated to be applicable to varied computer systems.

1.2 Contributions

This dissertation will first introduce techniques that eliminate barriers to parallelization so that performance is limited by true data dependences. Then, we further focus on reducing the cost of true data dependences to make real-world programs scale to large multicore systems. Throughout this dissertation, we focus on systematic and automated techniques to parallelize code pervasively throughout whole programs, with little or no assistance from the application programmer. We will use hardware support to preserve the semantics of sequential execution despite running tasks in parallel, preserving the ease of writing and debugging programs for sequential execution. We demonstrate the effectiveness and generality of parallelization techniques adhering to these principles by implementing them in three compiler-hardware systems: T4, S5, and ASH.

Chapter 3 – T4: We first introduce T4 [260], a compiler-hardware co-design for parallelizing whole sequential C/C++ programs. T4 strictly preserves sequential order in all programs, without requiring users to perform manual synchronization. Thus, T4 users will never suffer deadlocks, data races, or the nondeterministic ordering of concurrent tasks. T4 automatically leverages the novel Swarm CPU architecture (Section 2.3) to achieve four novel capabilities:

1. Exploit out-of-order spawns to organize tasks into rapidly expanding

trees, which perform high-throughput task spawns to keep cores busy without overburdening the system with idle tasks.

2. Eliminate false dependences on local variables and stack memory allocations, so parallelism is limited only by true data dependences.
3. Use dynamic information to exploit data locality for greater scalability. To our knowledge, T4 is the first parallelizing compiler that automates locality-aware execution for arbitrary—even pointer-based—data structures, without allocating extra storage.
4. Reduce parallelization overheads to make it profitable to create tiny tasks that unlock greater parallelism.

T4 crucially attains these capabilities across arbitrary loop nests, function calls, and complex control flow, unlocking parallelism without unduly restricting programmers. T4 is the first system for automatic parallelization that can achieve unbounded fine-grained parallelism on loops whose number of iterations is unknown. By implementing T4 as a series of code transformation passes within the industrial-strength LLVM/Clang compiler framework, we can scale complex real-world programs from SPEC CPU 2006 to up to 64 cores and achieve order-of-magnitude speedups.

Chapter 4 – S5: We introduce S5, which builds on T4 by providing abstract data types that can be incorporated into a C++ program to reduce gratuitous true data dependences, removing remaining limits on T4’s parallelism. S5’s key insight is that many data structures serve as *schedule structures* whose job is to enable deferred computation. Common implementations of these structures have frequent true data dependences that serialize operations, but this serialization is not needed by the application.

S5 addresses this problem with a library of abstract data types that provide freedom for the system to implement schedule structures using new techniques that minimize unneeded dependences. Under the hood, S5 exploits hardware and compiler support for task scheduling with fast dedicated hardware queues to avoid touching shared memory when the collection of tasks is small. Additionally, S5 introduces a new flexible hardware architecture that lets software define specialized task scheduling semantics tailored to the needs of an application to maximize efficiency when task collections grow large.

S5 scales *sequential* programs with unpredictable memory accesses linearly to hundreds of cores. These high speedups are achieved both

for a large existing sequential program, as well as for several graph algorithms that are challenging to parallelize, where S5 matches and sometimes far surpasses the performance of expert-optimized parallel Swarm programs. While S5 is already easy for a C++ programmer to use manually, we further ease adoption by automatically generating S5 code with UGC [34], a domain-specific compiler for graph analytics.

Chapter 5 – ASH: Finally, we demonstrate that these techniques continue to apply in a vastly different domain by presenting *ASH*, an accelerator specialized for digital circuit simulation. Specifically, ASH parallelizes Verilog and SystemVerilog simulations using a novel dataflow hardware architecture with a simple partitioned memory system and no cache coherence. ASH’s compiler conveys static information about memory accesses and data dependences to hardware, which enforces these dependences using novel prioritized dataflow mechanisms.

ASH uses merely 115 mm² of silicon area to provide 32× better performance than a state-of-the-art parallel simulator running on a large multicore CPU. ASH builds on Verilator, a very different compiler infrastructure from T4/LLVM, and reuses many of the same ideas as T4 and S5, such as conveying task ordering from software to hardware, generating task trees for parallel task spawns, minimizing parallelization overheads by efficiently passing values through registers, and exploiting hardware queues to decouple task spawn and execution.

* * *

T4, S5, and ASH demonstrate that automatic parallelization can be practical and effective if we have the right hardware-software interfaces. All three systems feature working compiler implementations that have been evaluated with large real-world workloads, not synthetic microbenchmarks. This provides evidence for the broad benefits of our new hardware and compiler techniques. Exciting future work remains in extending the capabilities of our new systems to make them even more scalable, general, and efficient (Chapter 6).

2 Background and Related Work

This chapter presents an overview of existing parallelization techniques that later chapters build on. We focus on reviewing general topics that are related to all chapters of this dissertation. Later chapters contain further background and prior work that is closely related to the specific systems presented in those chapters.

2.1 General Approaches to Parallelization

A critical challenge in parallelizing a program is to ensure execution order respects dependences among tasks. Serial execution lacks this challenge, as the sequential execution order can implicitly enforce dependences without explicitly tracking them during program execution. If all tasks, dependences, and latencies in a program are known in advance of program execution, a compiler could address this challenge by producing a fixed schedule of parallel execution that respects dependences [202, 203]. However, in most programs, input data is processed to determine what additional data to access and what control-flow paths to execute. Thus, a general system for pervasive parallelization must make dynamic decisions about when to execute tasks as they are discovered at runtime.

Dataflow (or *data-driven*) execution [55] is a conservative approach to parallelization that ensures tasks run in parallel only after their dependences are satisfied. In a dataflow program, a task must send a message called a *token* to every other task that depends on it. Tokens may optionally contain data to be transferred to the recipient task.¹ The system executes each task only after it has received every token

¹ Some authors define “true dataflow” architectures as those where each token carries data for a true data dependence. But architectures for dataflow execution of complex programs also use tokens for write-after-read and write-after-write dependences (e.g., due to dynamic memory reallocation for function activation frames) [163].

it will ever be sent. Any data in the tokens are provided as input to the recipient task, which executes either a single instruction [55, 75, 163, 165, 166, 176, 230] or a sequence of instructions [46, 64, 71, 157, 200, 261].

Dataflow execution maximizes parallelism when dependences are known before tasks execute, but dataflow-style dependence tracking also has drawbacks. Tokens that carry copies of data directly to recipients can improve parallelism, as they avoid the false dependences that would occur if tasks communicated by reading and overwriting variables in shared memory. Unfortunately, tokens waiting to be consumed incur storage costs, and tokens must be either dynamically matched to recipients using expensive associative lookups [163, 176, 230] or routed to recipient tasks that are pinned in advance to distinct hardware elements [40, 55, 75]. Additionally, dataflow systems must determine the tokens any task will receive before it runs. If a task might be dependent on many other tasks (e.g., because its execution might use one of many data elements) then it must receive tokens for every *potential* dependence. Dataflow execution performs poorly when tasks have many times more potential dependences than actual dependences. For example, in a graph-traversal algorithm such as breadth-first search, a task that visits one vertex in the graph may need to access the state of neighboring vertices. The actual data dependences caused by writes to vertices' state are not known until the tasks run and find their neighbors. In a dataflow execution, potentially dependent tasks would send tokens to each other in a serial fashion, even when many tasks are likely to be independent.

Most software parallelization techniques execute tasks in a partially ordered fashion to reduce the expense of tracking and enforcing dependences. Specifically, many parallel programs use *bulk-synchronous* parallelism [245], which organizes tasks into an ordered sequence of batches and prevents any overlapping of execution between one batch of tasks and the next. Synchronizing large batches of tasks in bulk (e.g., using explicit barriers, or a runtime system that finishes one parallelized loop or kernel before starting the next) may have lower performance overheads and simpler system requirements than a dataflow execution that sends separate tokens from individual tasks to each other. But bulk synchronization squanders opportunities to compose parallelism across batches, as serialization occurs between any two task in different batches even if the individual tasks are known to be independent. Thus, bulk synchronization is beneficial only in applications with copious parallelism, with large batches of unordered tasks.

Speculative parallelization is an alternative approach that offers more flexibility to execute tasks in parallel before their dependences are known.² With speculative parallelization, the compiler or programmer divides code into tasks that are likely to be independent. At runtime, the system tries to run these likely-independent tasks in parallel, detecting some dependences among them on the fly. Specifically, we consider speculation on the presence of dependences (rather than predicting data values [57, 138, 184]). The system must detect dependences that *conflict* with the order in which tasks executed. A conflict indicates a task *misspeculated*, meaning it ran too early. The conflicts must be resolved by *aborting* the task that misspeculated. An abort undoes misspeculated side effects such as memory writes. An aborted task may be re-executed at a later time when the previously conflicting dependence is now respected.

Speculation requires extra storage for *speculative state*, which facilitates detecting conflicts and undoing speculative actions, but the storage costs of speculation can be limited. A task’s speculative state is deallocated when it *commits*, which occurs when the system determines no further conflict could arise between this task and logically earlier tasks [185, 225]. Thus, the system holds speculative state and detects conflicts only within a limited window of uncommitted tasks. The storage for this speculation window can be sized proportionally to the number of cores in the machine, to hold enough tasks to keep cores busy.

The flexibility of speculation has several benefits for parallelization. By eliminating the need for tasks to know their dependences before they run in parallel, speculation makes it possible to parallelize more

2 This dissertation develops techniques to use many cores, which are orthogonal to core microarchitecture. But it is interesting to note the same concepts drive how core microarchitectures exploit instruction-level parallelism (ILP). CPU cores restrict ILP to a small window of instructions with their re-order buffers [214]. Thus, values destined to reach future instructions outside the window may be communicated (e.g., through a register file or ordinary loads and stores) without explicit dependence tracking. CPU cores with register renaming [176, 240] combine a restricted form of dataflow with speculative execution to achieve affordable storage costs. Specifically, they use dataflow-triggered execution only to enforce data dependences on registers, as these dependences can be determined before execution. Other dependences that arise due to data-dependent memory accesses or control flow are handled by speculation. The small number of registers in any ISA helps limit the size of storage structures (e.g., a physical register file and issue queue) needed to match in-flight register updates and recipient instructions for dataflow execution.

types of tasks, such as those that visit nodes in a graph traversal. Additionally, systems using speculation can identify when tasks incur actual dependences by observing actual data accesses at runtime, resulting in less serialization than conservatively enforcing potential dependences for dataflow execution or barriers between batches of tasks [147]. Finally, the flexibility of speculation allows some systems to enforce stronger guarantees in their execution models, which may make parallel software easier to write [93].

Choosing when to use speculation should be driven by a performance tradeoff: speculation gives more flexible parallel execution, but aborts and re-executions could hurt performance more than waiting to execute tasks when they are safe. Speculation is profitable if many tasks turn out to be independent so aborts are not too frequent. System implementation details can have a large effect on this tradeoff. This dissertation uses careful hardware-software co-design to reduce the costs of aborts and to uncover plentiful parallelism even in challenging programs. As a result, aggressive speculation becomes enormously profitable.

2.2 Parallelizing Compilers

Automating the parallelization of sequential code is a long-standing problem for which many techniques have been tried. We now provide a survey of work in automatic parallelization, which uses a compiler to translate sequential code to run in parallel. We will first discuss non-speculative parallelization, which achieves high scalability and efficiency for some loops, but fails to apply to many real-world programs. We then discuss work in speculative parallelization in software, where advanced techniques can significantly benefit some specific workloads. Finally, we review prior work that has used hardware support to extract parallelism from more challenging applications. This dissertation, in part, shares the goals of this prior work, but takes a different approach: dynamically scheduling tasks to exploit fine-grain parallelism and locality with mechanisms that avoid serial bottlenecks.

2.2.1 Non-speculative parallelization

Non-speculative parallelizing compilers [31, 154] transform code within a sequential program to create tasks that are safe to run in parallel. These compilers can succeed only when static analyses ensure at compile time

that different tasks executing a block of code will be independent at runtime. A critical enabling transformation that parallelizing compilers perform is to eliminate false dependences using *privatization*, which allocates extra copies of a variable to be used by parallel tasks [31, 149, 247]. Privatization is safe when static analyses can prove the privatized variable had no true data dependences between tasks, that is, it holds a temporary value consumed only within the task where the value is produced. Polyhedral compilers [32, 79] can parallelize some iterations within loop nests that traverse dense tensors (i.e., multidimensional arrays) with regular, fixed-stride access patterns. But many programs are *irregular*: they make frequent use of data-dependent control flow and unpredictable memory accesses (e.g., due to multiple indirection), preventing static analysis from identifying or creating independent tasks. [94, 114].

For irregular programs, compilers and hardware for non-speculative parallelization has focused on specialized forms of dataflow execution to exploit *pipeline parallelism* in inner loops. DSWP [170, 191] and HELIX-RC [44] use static analysis to partition the body of a loop into *stages* in a feed-forward pipeline. A single core executes each pipeline stage, localizing loop-carried dependences. All other dependences are handled with hardware support for efficient unidirectional messaging from one core to the next. These pipelines cannot parallelize stages with cyclic dependences. More recently, Pipette [161] adds hardware support for time-division multiplexing to handle load variation among stages as well as additional hardware support for memory indirection, enabling the Phloem compiler to decouple a loop or loop nest into a larger number of pipeline stages to improve efficiency [162]. In some cases, DSWP+ [97] and Pipette/Phloem can also parallelize a stage by identifying independent tasks (from different loop iterations) within that stage that are safe to run on multiple cores. If a parallelized stage writes to an array, each core shall own a partition of the array and the owning core must execute all tasks accessing its partition to avoid data races and cache ping-ponging [85].

Ultimately, while non-speculative approaches to automatic parallelization can make some computation kernels fast, they serially execute any code outside the loops they parallelize and do not compose parallelism across across multiple invocations of a parallelized loop nest [96].

2.2.2 Speculative parallelization in software

Like in this dissertation, some prior compilers leverage speculation to parallelize a broader range of sequential code. But software-only speculative parallelization [11, 97, 110, 148, 190, 192] incurs significant overheads, especially to recover from misspeculation, so it is profitable only for kernels where dependences that cause aborts are extremely rare.

Some compilers exploit application properties to reduce the number of dependences and aborts. Speculative privatization is a compiler technique that eliminates some false dependences at the cost of increased memory usage and run-time checks that validate the safety of data accesses [111, 192]. Recent work uses profiling and static analysis to reduce the overheads of these run-time checks, but still suffers from expensive misspeculation recovery that makes it unprofitable for applications with frequent conflicts [11].

2.2.3 Speculative parallelization with hardware support

Thread-level speculation (TLS) architectures propose to use hardware mechanisms for efficient conflict detection and recovery to make speculative parallelization more broadly beneficial for sequential programs [63, 72, 83, 193, 194, 218, 226, 269]. Unfortunately, TLS architectures suffer from expensive unselective aborts and serial spawn and commit mechanisms. This is because they chose to use simple hardware that tracked task order by organizing all tasks into a linear chain, so hardware could simply traverse the chain to perform in-order commits or find possibly dependent tasks to abort. As Chapter 3 will show, this prevents scaling tiny tasks to many cores.

TLS compilers [28, 48, 57, 112, 171, 187, 242, 248, 263] are limited by the architectures they target. Studies on selecting tasks for the earliest TLS architectures consider function calls and loop iterations [169, 248]. However, without sufficient hardware support for tiny tasks, previous compilers have focused on *selectively* parallelizing coarser tasks. Many have focused on techniques limiting the use of speculation to avoid the significant cost of unselective aborts imposed by their hardware. Du et al. propose models to statically estimate the likelihood of data dependences, to avoid parallelizing loops that could yield frequent aborts [57]. Others found more success using profiling to select loops for parallelization

where aborts are rare, although this requires a more complex compilation process where a programmer must supply representative inputs [44, 66, 190]. POSH [142] was the most recent compiler attempting to parallelize whole C/C++ programs, considering both function calls and loop iterations as task boundaries. POSH still relied on profiling to choose when *not* to spawn a task at a potential task boundary, preferring to form sufficiently coarse tasks to amortize per-task overheads and avoid likely aborts.

Many TLS compiler techniques used hardware that did not support nested spawns, and therefore were restricted to choosing a single loop level to parallelize in each loop nest. Some prior work has developed specialized synchronization techniques exploit the assumption that only a single loop is parallelized so tasks may be statically distributed to cores in a rigid fashion. Zhai et al. used compiler techniques to selectively synchronize common classes of data dependences instead of speculating on them [263, 264]. This dataflow-style synchronization forms a long *critical forwarding path* that includes every iteration of a parallelized loop that limits parallelism if each iteration is short. SpecDSWP [244] addresses the latency sensitivity of this critical forwarding path by pinning loop fragments to cores such that the frequent inter-core dependences form a feed-forward pipeline. This hides communication latencies after the pipeline fills and reaches a steady-state, but prevents using more cores than there are pipeline stages. These rigid scheduling schemes have some benefits and significant drawbacks. On the one hand, for applications dominated by inner loops with dependent iterations, a rigid schedule makes synchronization simple, and it ensures frequent dependences are safely respected, avoiding aborts. On the other hand, a rigid schedule stalls cores whenever there is load imbalance [171], so cores spend time idle even in steady state with no aborts. Furthermore, it restricts parallelism and does not allow for nested task spawns so it slow to fill a machine with work when the loop starts or resumes execution after an abort. By contrast, this dissertation seeks to serialize only dependent tasks, running other tasks whenever available to keep tens or hundreds of cores busy. Moreover, we always parallelize a whole program, composing parallelism across loop nests and functions calls. Finally, we uncover asymptotically more parallelism by parallelizing task spawning itself, which delivers order-of-magnitude speedups by quickly filling the machine with work.

2.3 Swarm

The systems presented in this dissertation build upon the Swarm architecture [107, 108]. Swarm extends a multicore with hardware support for speculative tasks. We use Swarm because it scales tasks as short as tens of instructions to hundreds of cores [106, 109, 227]. Below, we provide a unified description of Swarm hardware that combines techniques that were published separately as they were developed over several years [1, 2, 3, 106, 107, 108, 109, 227].

2.3.1 Application-level execution model

A Swarm program consists of tasks whose execution is managed by hardware. When a task runs, it may read and write shared memory, create children tasks using a `spawn_task` instruction when it finds more work to do, and perform arbitrary actions determined by the application, even making system calls. Each task ends by running the `dequeue_task` instruction, which jumps to the start of the next available task.

When each task is created, software specifies the task’s priority by assigning an integer *timestamp* that conveys the order in which tasks should appear to run. This dissertation also uses Fractal [227], which modifies priorities by placing each task in a hierarchy of nested *domains*. A program’s first task starts in an initial *root domain*. Within a domain, tasks have timestamp-ordered semantics, with lower timestamps coming first.³ A task that spawns children tasks to the same domain must assign them timestamps equal to or greater than its own. Tasks appear to run in increasing timestamp order, as if a single core dequeues one task at a time from a min-priority queue. These semantics match those of the initial Swarm architecture without Fractal.

Fractal provides strong atomicity⁴ guarantees across domains to allow parallel composition of speculative algorithms. Each task can create a single new subdomain and spawn tasks into it. These subdomain tasks may create their own further nested subdomains. All tasks in a domain appear to execute after the task that creates the domain and

³ Tasks do not need to execute at consecutive timestamp values without gaps. This makes timestamps different from sequence numbers used to ensure ordering in network protocols, storage systems, and some dataflow architectures [230].

⁴ We use terminology from computer architecture literature, where a group of operations is *atomic* if the group will never appear to be interleaved with another concurrent operation. This is called “serializability” [174] in the databases literature.

are not interleaved with other tasks outside the domain. In other words, any non-root domain together with its creator appears to execute as a *single atomic unit* in isolation within the parent domain. Tasks may also spawn tasks to their parent domain. This lets a task delegate spawning future tasks to descendants within the domain it creates. A task cannot spawn children to any domain beyond the domain it belongs to, its parent domain, and the single domain it may create.

2.3.2 Swarm programming example

Parallel Swarm programs may be written manually through a low-level C++ interface. Documentation [259] and many full Swarm example programs are available online, within the Swarm simulator code repository [231]. Listing 2.1 illustrates some key features of the low-level Swarm interface by showing a simple manually parallelized implementation of the maximal independent set algorithm, which selects a subset of the vertices in a graph. Further discussion of this algorithm and its parallelization strategies may be found in the literature [30, 227].

Each task executes a C++ function that takes a timestamp and arbitrarily many additional arguments. Tasks do not return values nor pass references to local variables (i.e., stack allocations) to other tasks. Thus, tasks may only communicate through arguments passed by value, statically allocated (i.e., global) variables, references to shared heap allocations, and I/O. Tasks create children tasks by passing a function pointer, timestamp, and any additional arguments to `swarm::spawn()`, which translates to a `spawn_task` instruction. By default, `swarm::spawn()` places the child task in the same domain as its parent, but it takes optional `PARENTDOMAIN` and `SUBDOMAIN` flags to place a child in other domains. `swarm::spawn()` optionally takes a spatial hint [106] as its third argument, which is an integer that abstractly denotes the data the task is likely to access. As we will discuss later, hints aid the architecture in performing locality-aware execution to reduce data movement and aborts.

Listing 2.1 defines three task functions.⁵ The initial task runs `computeMIS()`, which spawns tasks to run `tryInclude()` for every vertex. When `tryInclude()` runs, it checks whether its vertex has already been visited. If it has not, it adds that vertex to the independent set and

⁵ For simplicity, these tasks use serial **for** loops to spawn their children. Chapter 3 will discuss more complex techniques to spawn tasks in parallel.

```

1 static inline uint64_t cacheLineHint(void* addr) {
2     return static_cast<uint64_t>(addr) / BYTES_PER_CACHE_LINE;
3 }
4
5 static void exclude(swarm::Timestamp, Vertex* v) {
6     v->state = EXCLUDED;
7 }
8
9 static void tryInclude(swarm::Timestamp ts, Vertex* v) {
10     if (v->state == UNVISITED) {
11         v->state = INCLUDED;
12         for (Vertex* ngh : getNeighbors(v)) {
13             swarm::spawn(excludeNode, /*timestamp=*/ts,
14                         cacheLineHint(ngh), /*additional arg=*/ngh);
15         }
16     }
17 }
18
19 void computeMIS(swarm::Timestamp, Vertex* vertices, size_t numVertices) {
20     swarm::createDomain();
21     for (size_t i = 0; i < numVertices; i++) {
22         Vertex* v = &(vertices[i]);
23         swarm::spawn<SUBDOMAIN>(tryInclude, /*timestamp=*/i,
24                               cacheLineHint(v), /*additional arg=*/v);
25     }
26 }

```

Listing 2.1: A manually parallelized Swarm algorithm for computing a maximal independent set of vertices in a graph.

spawns a task to run `exclude()` for each neighboring vertex. `exclude()` marks a vertex as excluded from the independent set. `tryInclude()` and `exclude()` tasks both have the cache line address they will access as their spatial hint and take a vertex pointer as an additional argument.

Manually writing Swarm programs requires taking great care with timestamps and domains to ensure tasks are ordered correctly. In Listing 2.1, `computeMIS()` calls `swarm::createDomain()` to create a new domain where all other tasks in the algorithm will run, making the algorithm an atomic unit that is safe to use in a larger application program. `computeMIS()` also assigns a distinct timestamp to each task that runs `tryInclude()`, ensuring vertices are added to the set in a deterministic order matching the behavior of typical sequential implementations. Finally, `tryInclude()` spawns `exclude()` tasks with the same timestamp as their parent. As long as the tasks that include

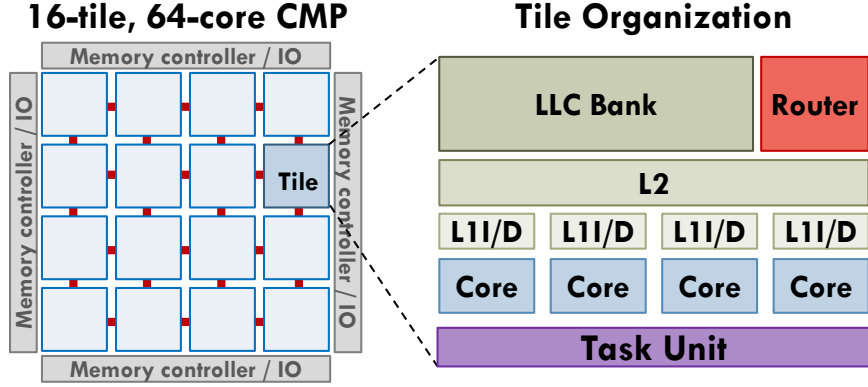


Figure 2-1: 64-core/16-tile Swarm processor configuration.

one vertex and exclude its neighbors share a timestamp value distinct from all other tasks, they will not appear to be interleaved with other tasks, which is crucial to the correctness of the algorithm’s output. Under the hood, Swarm may execute tasks from different timestamps and domains in parallel as long as they are independent.

2.3.3 Swarm microarchitecture

Swarm makes modest changes to a tiled, cache-coherent multicore CPU. We assume each tile in our baseline CPU has a cluster of cores, each with its own private L1 caches. The cores in a tile share a tile-private L2 cache, and each tile has a slice of of a shared last-level cache (LLC).

Swarm runs tasks speculatively and out-of-order to exploit parallelism. We focus on Swarm’s key distributed hardware mechanisms that this dissertation exploits or adapts to achieve good performance. (Further hardware implementation details can be found by following the provided citations to prior work.) Swarm efficiently supports fine-grained tasks and a large speculation window by combining four features: hardware task management with large queues, distributed mechanisms for tracking a sequential order, scalable high-throughput speculation, and locality-aware execution.

Hardware task management and queues: Rather than a centralized priority queue, Swarm has distributed *task units* that queue tasks near cores. Specifically, Swarm has one task unit in each four-core tile, as shown in Figure 2-1. These task units add <2% area overhead to a standard multicore [107].

When a core runs the `spawn_task` instruction, hardware collects

the instruction’s operands to form a *task descriptor* that contains the child task’s timestamp, function pointer, additional arguments, and other metadata. This descriptor is buffered in the local task unit’s *task send buffer (TSB)*. `spawn_task`’s operands are taken directly from registers, not memory, making task creation cheap so it requires the involvement of the core for just a few clock cycles. Subsequently, the TSB asynchronously sends descriptors over the on-chip network using point-to-point messages, without involving cores or any central scheduler. A *task queue (TQ)* in each tile’s task unit receives the descriptors.

When a core becomes available, the TQ can dispatch any available task to the core to begin speculative execution, with the arguments from the task descriptor made available in general-purpose registers. Each TQ prioritizes its tasks based on program order (timestamps), not spawn order, making it profitable to spawn tasks far in advance. This prioritization is implemented with a fixed-capacity pipelined heap [2, 27, 101], which allows inserting and dequeuing a task every cycle in priority order.

When a core finishes a task, the task unit holds the task’s speculative state in a *commit queue (CQ)* until the task commits. The core is then immediately available to execute another task. CQs are large (e.g., 16 tasks per core), so cores can run many tasks ahead of commit. Thus, if its local TQ has tasks ready to run, each core keeps executing these tasks without waiting for cross-chip communication to see if previous tasks can commit. However, filling the system with tiny tasks is challenging. This dissertation develops techniques to provide high-throughput task spawn and sufficient load balance to keep tens or hundreds of cores busy.

Distributed tracking of sequential order: To enforce atomicity of equal-timestamp tasks, hardware adds information to each task’s timestamp to give each task a unique *virtual time (VT)*. This enables any two tasks’ VTs to be compared to determine their relative order. Swarm lazily constructs VTs after task dispatch by appending a unique integer *tiebreaker* [3] to each task’s timestamp, producing in a wider integer called a *domain virtual time (DVT)*. Each task in the root domain (including all tasks in programs that do not use Fractal) simply uses this DVT as its VT.

VTs also enforce the Fractal execution model’s strong atomicity guarantees across domains. To do this, tasks in non-root domains have VTs formed by appending their unique DVT to the VT of the task

that created their domain. Thus, each VT is a sequence of DVTs. These sequences are ordered lexicographically. VTs may grow arbitrarily long if an application creates deeply nested domains, but a hardware optimization called *zooming* [227] virtualizes long VTs so that most operations are performed cheaply, with the same hardware cost as fixed-bitwidth VTs. When needed, hardware “zooms in” to a domain to provide finer VT resolution by temporarily dropping the shared common prefix of VTs in that domain so each task stores a shorter remainder of its VT. Later, the system must “zoom out” to continue executing tasks outside that domain.

Scalable high-throughput speculation: Swarm extends prior mechanisms for data dependence speculation to scale to a large multicore system. Specifically, Swarm builds on prior techniques that use caches for eager (undo-log-based) version management with eager (coherence-based) conflict detection at cache-line granularity, using Bloom filters like LogTM-SE [257]. Swarm improves the scalability of eager conflict detection by leveraging the cache hierarchy to reduce conflict checks and to ensure most conflict checks are performed locally within a tile, avoiding frequent global communication [107].

Unlike early HTMs like LogTM-SE but like most TLS architectures, Swarm allows later tasks to read a speculatively updated value before the update commits, a feature known as *speculative forwarding* [102, 182, 186, 188, 218, 226]. This is critical for parallelizing ordered tasks that may take a long time to commit. The cost of forwarding is that, if a task aborts, the system must abort data-dependent tasks to which it forwarded data. Swarm accomplishes this by reusing coherence-based conflict detection to discover and recursively abort data-dependent tasks during the rollback of memory writes. Swarm selective aborts only descendants and data-dependent tasks. As we will see, this presents opportunities to reduce abort costs by avoiding needless ancestor-descendant relations and data dependences.

To drain the CQs, Swarm adopts the virtual-time algorithm from Time Warp [105] for its high throughput with low communication costs. Periodically (e.g., every 100 cycles) the task units perform a hierarchical min-reduction to determine the lowest VT associated with an unfinished task. This reduction is implemented by task units communicating through the network over tens of cycles. All lower-VT tasks then commit in bulk. This scheme allows many tasks to commit per cycle [2]. Because tasks may run far out of order, a CQ may sometimes fill up.

To ensure forward progress, a full CQ may abort higher-VT tasks to free CQ entries needed for late-arriving lower-VT task to run [107].

Locality-aware execution: To exploit locality, each task can optionally be given a *spatial hint*, an integer that denotes a memory location that the task is likely to access. This hint is an operand to the `spawn_task` instruction. Hardware sends same-hint tasks to run in the same tile, by hashing each hint to get a destination tile ID [106]. Within a tile, hardware serializes same-hint tasks to avoid likely conflicts.

2.3.4 Supporting system features

Swarm provides additional features not used directly by application programmers, but needed to enable scalable and efficient implementations of system libraries.

Precise exceptions: Swarm guarantees apparent sequential execution even when tasks incur exceptions. Exceptions (e.g., due to segfaults or system calls to perform I/O) require special handling because they can perform *irrevocable actions*, meaning hardware cannot undo their effects. Swarm can execute these irrevocable actions safely by first aborting any speculative task that attempt them, then waiting for the task to become the lowest-unfinished-VT task before allowing it to run non-speculatively [109]. This process is slow as it serializes tasks that perform irrevocable actions, but it is used rarely.

Capsules: Capsules [109] are a general solution for providing protected access to software-managed speculation. Similar to but more general than escape actions or transactional suspend in transactional memory systems [43, 158, 272], Capsules can avoid the overheads of hardware conflict detection, perform parallel irrevocable actions, and safely undo speculative actions in software. Capsules provides three hardware mechanisms to achieve these capabilities. First, the dynamic linker and loader uses a privileged interface to specify a set of thread-safe *capsule functions*, which may bypass hardware speculation to perform irrevocable actions such as system calls without serialization. Second, capsule functions can access *untracked memory* regions protected from corruption by misspeculating tasks by virtual-memory mechanisms. Untracked memory is neither conflict-checked nor versioned in hardware, eliminating speculation overheads for accesses to untracked data. The capsule function may use conventional synchronization (e.g., locks) to ensure accesses to untracked data is safe. Third, capsule functions

can register an abort handler, a software routine hardware may run non-speculatively to undo the capsule’s effects in untracked memory. The TSB holds the abort handler while its parent task is speculative. Upon the parent task’s commit, the abort handler is discarded. If the parent task aborts, the abort handler is sent to the local TQ which dispatches it to run non-speculatively on the next available core.

2.3.5 Handling unbounded tasks with finite hardware queues

Hardware protection of untracked memory is used not only for capsules, but also for software to safely virtualize Swarm’s TQs to enable applications to create unboundedly many tasks and Fractal domains. During the startup of every Swarm program, the runtime uses special hardware instructions to register two software routines called *spillers*. Spillers take tasks that will not run for a while from the TQ and store them to memory. This process frees hardware resources, thereby preventing deadlock and reducing full-queue stalls.

One spiller handles overflowing TQs. Whenever a TQ is nearly full, before dequeuing its next regular task, the TQ first dispatches this spiller to run on the next available core. The spiller runs in a privileged non-speculative mode where it may execute a `spill_task` instruction to remove the highest-timestamp available task from the TQ. This instruction returns the contents of a task descriptor in registers, which the spiller writes to a chunk of untracked heap memory. After spilling several (up to 15 in our implementation) descriptors into the untracked memory chunk, the spiller spawns a single *requeuer* task that will respawn the spilled tasks.⁶

Requeuers themselves are spilled recursively, creating an in-memory priority queue. Each requeuer has an argument pointing to the spilled task descriptors it will restore, and if the requeuer is spilled, this pointer will be stored in memory as part of the requeuer’s own descriptor. Thus, the spilled descriptors are organized in untracked memory as a forest of 15-ary trees, and the TQ holds a requeuer at the root of each tree. We assign each requeuer a timestamp equal to the lowest timestamp among the spilled tasks it will restore, making each tree into a 15-ary min-heap

⁶ Previous papers have used “coalescer” in place of “spiller” and “splitter” in place of “requeuer”, indicating effects on TQ occupancy. The spiller “coalesces” multiple TQ entries by replacing them with a single TQ entry for the requeuer. The requeuer performs the inverse operation: “splitting” itself into the restored original tasks.

that restores lower-timestamp tasks before higher-timestamp ones.

To avoid complicating the implementation of aborts, we never spill a task whose parent has not committed [109]. This ensures an aborting parent task can find its children still present in hardware queues where they may be quickly aborted and discarded. To prevent unspillable tasks from deadlocking the system, we reserve some entries in each TQ and TSB that are only allowed to hold spillable tasks. Additionally, we reserves TQ entries before executing a requeuer to ensure the requeuer will not stall on a full queue when spawning tasks. Like spillers, requeuers always run non-speculatively and do not use commit queue entries. This is safe because they cannot misspeculate: they do not access tracked memory, they only observe non-speculative state in untracked memory, and they are not spawned by speculative parent tasks.

For applications that create tasks at enough domain depths to require Fractal zooming, there is also a spiller that removes TQ entries whose VTs can no longer be represented after zooming in [227]. This spiller is very similar to the TQ-overflow spiller we just described, with the difference being that it places its child requeuer in the subdomain that is being zoomed into, and the requeuer can run to respawn the spilled tasks to the parent domain only when the system has zoomed back out.

3 T4: Trees of Tiny Timestamped Tasks

3.1 Introduction

Multicore systems have been ubiquitous for years, but programmers still write sequential code. Parallel programming remains a specialized skill, with pitfalls such as deadlocks, data races, and non-determinism [134]. Although some applications feature computations that are easy to parallelize, many programs feature data-dependent branches and multiple levels of memory indirection. While these features improve productivity, they stymie the parallelization of sequential code, as programmers and compilers cannot reliably determine what work is independent and thus safe to run in parallel.

Parallelizing programs where dependences are not known in advance requires *speculative parallelization*. With speculative parallelization, the compiler divides the sequential code into *tasks* that are likely to be independent. At runtime, the system tries to run these likely-independent tasks in parallel, detecting dependences among them on the fly. Dependences cause some tasks to be aborted and re-executed to preserve sequential behavior. Speculative parallelization can be done efficiently with hardware support, by reusing existing mechanisms (caches for version management and cache coherence to detect dependences).

Unfortunately, prior compilers and architectures for speculative parallelization of sequential code, known as thread-level speculation (TLS), have proven highly profitable only in limited cases, and achieved little speedup on many real-world applications [63, 72, 83, 193, 194, 218, 225, 244, 269]. TLS architectures suffered from four shortfalls that limited scalability:

1. Resolving dependence conflicts by aborting all later tasks *en masse*,

making aborts very expensive.

2. One-task-at-a-time spawn or commit mechanisms, which bottleneck parallelism among tiny tasks.
3. Lack of support for locality-aware execution.
4. Lack of support for scalable speculative memory allocation, resulting in false dependences (e.g., when parallel tasks share and reuse preallocated memory for stack frames).

Swarm (Section 2.3) features mechanisms that could address the scalability limitations of TLS hardware. But Swarm was designed for explicitly parallelized code, and adapting prior TLS compiler techniques Swarm yields poor performance. Swarm needs a new approach to automate parallelization. Without it, its utility is limited.

To address this challenge we present *T4*, a compiler that speculatively parallelizes sequential programs building on Swarm’s techniques. T4 stands for Trees of Tiny Timestamped Tasks, which summarizes how we tackle the limitations of TLS compilers. Specifically, T4 contributes four novel techniques to improve scalability:

1. T4 divides the program into *tiny tasks*. It breaks every function in the entire program into tasks of a few instructions each. By working at this fine granularity, T4 exposes parallelism aggressively and *isolates contentious memory accesses* to make aborts cheap.
2. T4 spawns tasks in parallel, forming *task trees*, not serial chains of spawns as in TLS. T4 assigns each task a *timestamp* to record program order while spawning tasks out of order. Trees of timestamped tasks expand in parallel to quickly fill large systems with work, and they enable selective aborts.
3. T4 *exploits dynamic information to improve locality* by sending tasks that access the same data to the same tile in the system, reducing data movement.
4. T4 spawns *tasks with few inputs*, not threads as in TLS. T4 carefully manages memory and register allocation to make task spawns very cheap, to avoid false dependences due to stack allocation, and to exploit hardware support within a scalable `malloc()` implementation.

We implement T4 as a series of compiler transformation passes within the LLVM [132] compiler framework (Section 3.3), resulting in a high-performance compiler that parallelizes existing C/C++ programs. We evaluate T4 with ten SPEC CPU2006 C/C++ programs, which have frequent dependences that limited the effectiveness of prior work (Section 3.10). T4 broadens the range of applications that benefit from

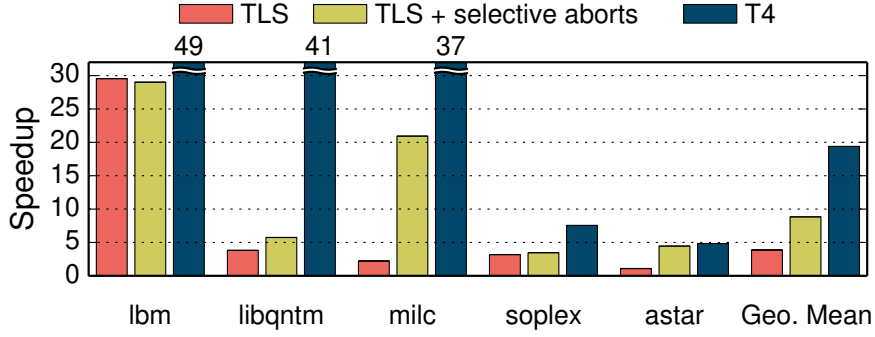


Figure 3-1: 64-core performance of T4 significantly exceeds a TLS compiler based on prior work. Speedups are relative to serial code compiled with `-O3`. More details are in Section 3.10.3.

speculative parallelization and yields large scalability improvements over prior compiler techniques on several challenging applications. Figure 3-1 compares the performance of T4 and a TLS system that combines many features from prior work, for five benchmarks running on 64-core Swarm. T4’s novel automatic program transformations contribute significant scalability. In short, *new compiler techniques are needed to exploit hardware for scalable speculative parallelization*.

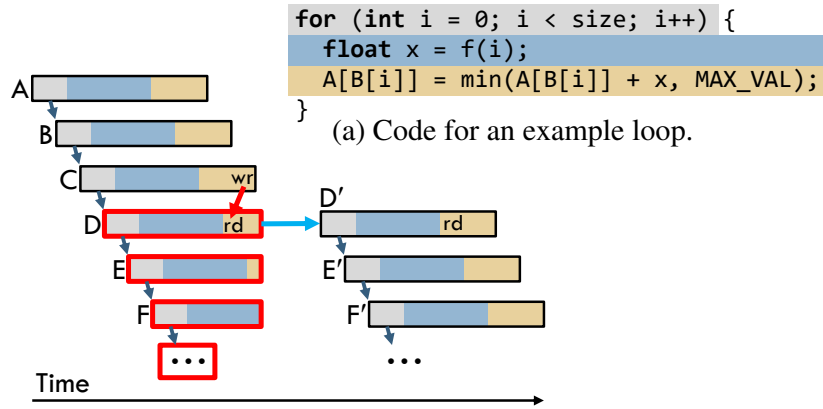
Our results show that, with sufficient compiler and hardware support, speculative parallelization can scale sequential programs to tens of cores, even with contentious memory access patterns. These results required modifying less than 0.1% of the original source code, and did not require any changes to interfaces, data structures, or the program’s sequential semantics. T4 lets programmers unlock parallelism while retaining the simplicity of sequential programming. T4 is open source and publicly available at <http://swarm.csail.mit.edu>.

3.2 Motivation

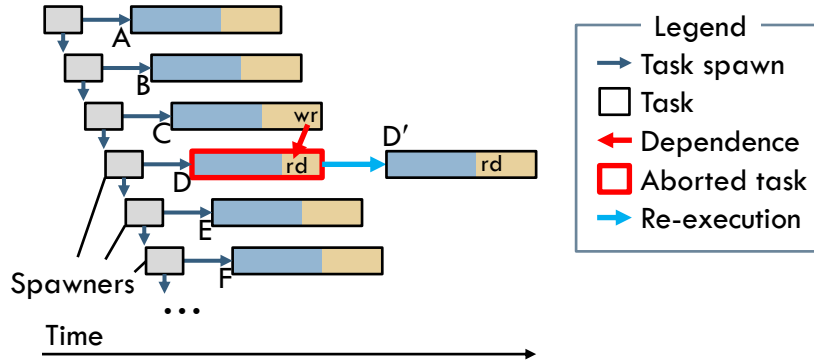
We now illustrate the limitations of prior TLS compilers and introduce key T4 features to overcome these limitations, explaining the large speedups in Figure 3-1.

3.2.1 Spawner tasks enable selective aborts

Figure 3-2(a) shows the code for an example loop that represents a common pattern found in many programs: each iteration first performs



(b) In a TLS task chain, aborting D also aborts later tasks E,F,...



(c) T4's spawners allow D to abort without aborting other tasks.

Figure 3-2: Execution timelines for speculatively parallelizing a loop, with each iteration constituting a task. A write in task C conflicts with a read of the same address in task D, resulting in aborts.

some work to compute a local value x , highlighted in **blue**, and then performs a read-modify-write memory operation on $A[B[i]]$, highlighted in **gold**.

Figure 3-2(b) illustrates how prior TLS systems could parallelize this loop: each task begins by spawning the next in the chain, and then runs an iteration of the loop. Tasks are spawned in program order, as required by most TLS architectures because of their simple mechanisms to enforce sequential semantics with in-order commits based on spawn order.

To preserve sequential semantics, the system tracks the memory

addresses accessed by each task, detecting *conflicts* when two accesses to the same location run in the wrong order and at least one access is a write. In Figure 3-2(b), task C writes to an address that conflicts with a read by task D. This conflict must be resolved by *aborting* task D, and re-executing it (as task D') so it is correctly ordered after task C. Any later-ordered task that received incorrect data from D must also abort. To implement this cascade of aborts, prior TLS architectures conservatively abort *all later tasks* en masse [72, 83, 194, 218, 225]. These unselective aborts wastefully discard work. They may also impede scalability by using global synchronization, stalling the execution of all later tasks until rollbacks have completed.

To address this problem, T4 leverages Swarm's *distributed selective aborts*, wherein an aborting task triggers additional aborts only for *dependent tasks* [107]. However, in Swarm and TLS systems that permit dynamic task spawn, a child task must abort if its parent aborts, as the parent may have created it due to misspeculation. To avoid cascading aborts, the compiler should avoid unnecessary parent-child relationships. The simplest way T4 accomplishes this is shown in Figure 3-2(c). A chain of **gray** *spawner* tasks decouples spawns from work: each spawner spawns the next spawner in the chain, and then spawns a separate *worker* task to execute the loop body. Furthermore, T4 ensures worker tasks avoid dependences due to the reuse of shared memory, e.g., for stack frames allocated for function calls. Now, D can abort without affecting later tasks that did not access the same data. Spawners neither access nor suffer conflicts on application data, avoiding unnecessary abort cascades. Figure 3-1 shows that selective aborts yields large speedups over TLS.

3.2.2 Tiny tasks make aborts cheap

The read-modify-writes of array A may cause aborts, but so far each abort forces re-executing an entire loop iteration, as shown in Figure 3-3a. This may waste a large amount of work, like the call to function f in our example.

To make aborts cheap, T4 spawns the call to f into a task in **blue**, separate from the contentious accesses to A in the call's continuation in **gold**. The new task boundary serves as a checkpoint, *isolating* the contentious accesses. A conflict now only requires aborting and re-executing a cheap tiny task, as shown in Figure 3-3b. If the function f

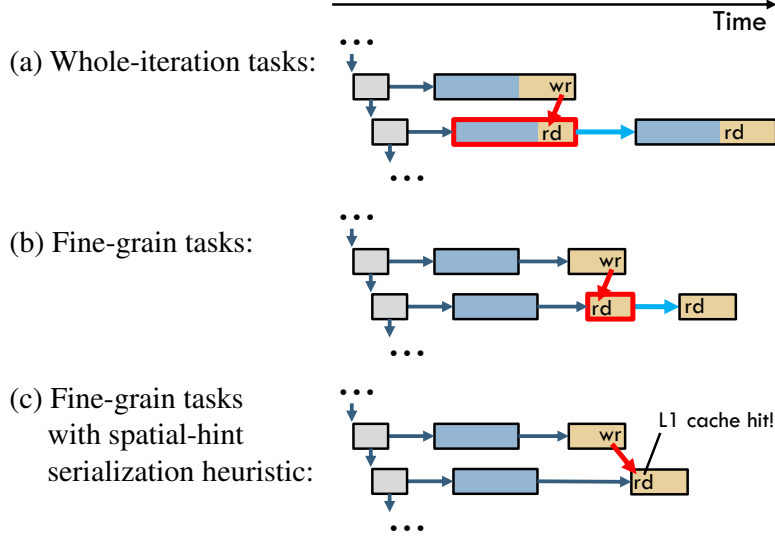


Figure 3-3: Execution timelines depicting a data dependence. Tiny tasks make aborts cheap and enable spatial hints.

contains loops or additional function calls, T4 would also split it into tiny tasks.

Splitting the tiny read-modify-write operations into tasks also allows an important optimization: when they are spawned, T4 can tag **gold** tasks with the cache line they will access as a *spatial hint*. Swarm hardware runs same-hint tasks at the same chip tile to exploit locality (Section 2.3). This prevents the indirectly accessed cache lines of array A from ping-ponging across the chip. Furthermore, Swarm serializes the execution of same-hint tasks, avoiding aborts altogether, as shown in Figure 3-3c. Thus, hints use dynamic information to decide which tasks run in parallel.

With the tiny tasks needed to make aborts cheap, achieving high parallelism requires hardware support. With only a few instructions, some tasks may be only tens of cycles, or shorter. To keep 64 cores busy, the system must be able to spawn, dispatch, and commit more than one task per cycle. Prior TLS systems could not achieve this throughput [107], either because they could not spawn tasks out of order, or they featured one-task-at-a-time commit mechanisms that bottlenecked throughput when using tiny tasks [194]. Consequently, they used static or profiling-based techniques to select coarse tasks, or merged tasks to make them coarse at runtime, to amortize the spawn and commit bottlenecks. These coarse tasks sacrificed parallelism

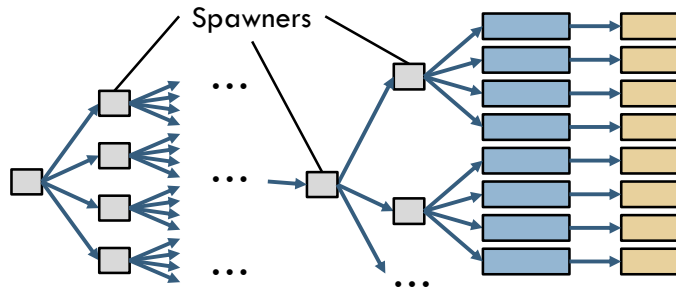


Figure 3-4: Exponential spawner trees improve parallelism.

and made aborts costly. Faced with enormously costly aborts, some prior systems limited their use of speculative parallelization to loops or program segments where dependences that would cause aborts were very rare [44, 66, 142, 190, 252].

T4’s ability to make aborts cheap allows aggressively speculating for parallelism, even in applications where conflicting memory accesses are common, such as `soplex` and `astar`.

3.2.3 Exponentially expanding trees make tiny tasks scale

Spawning tiny tasks with sufficient throughput to keep many cores busy also requires new strategies in software. The serial chain of spawners in Figure 3-2(c) cannot achieve high spawn throughput, because only one core is spawning tasks at a time. These serial task spawns bottleneck performance, limiting speedups to at most a few percent for the innermost loops in SPEC CPU2006 [117].

To address this issue, T4 transforms the code to use exponentially expanding trees of parallel spawners, as depicted in Figure 3-4. T4 introduces *progressive expansion* to produce spawner trees even for loops with unknown bounds or exit conditions (Section 3.6). When a loop starts executing, spawner trees rapidly expand to distribute the load of both spawners and workers across the system. Figure 3-1 shows that T4’s task trees offer further dramatic benefits in performance, on top of selective aborts.

3.3 T4 Implementation Overview

Table 3-1 summarizes how Swarm features (Section 2.3) couple with T4’s compiler techniques for effective speculative parallelization.

Design goal	Hardware mechanism	Compiler transform
Composable order	Timestamps & domains	Topological task ordering
Avoid dependences	Capsules for heap allocation	Stack elimination
Cheap aborts	Selective aborts	Spawners & tiny tasks
Cheap task spawn	Async. reg-to-reg communication	Task lifting & live-in reduction
High spawn throughput	Distributed task units	Progressive expansion
Exploit locality	Spatial hints	Spatial-hint generation

Table 3-1: To achieve its design goals, T4 introduces new software transformations to exploit hardware features.

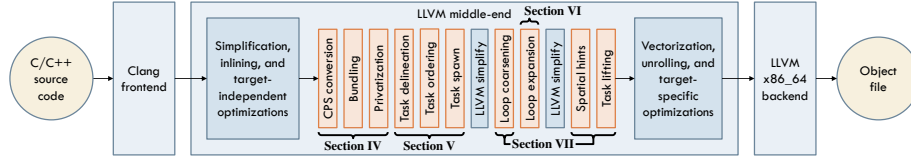


Figure 3-5: Structure of T4, with corresponding sections in this chapter. T4 is implemented by adding a series of passes to LLVM’s middle-end. Transformation passes newly implemented for T4 are highlighted in orange.

Swarm has previously been used to manually parallelize algorithms using programmer-assigned timestamps, e.g., to scale graph kernels that schedule tasks with a priority queue [107, 227]. Instead, T4 generates timestamps *implicitly* to preserve sequential semantics in a scalable way. T4 develops new techniques that extract parallelism despite irregular control flow, while maintaining the ease of writing deterministic sequential code and without changing sequential application algorithms or data structures.

Unlike prior work [44, 66, 142, 190, 252], T4 does not use profile-guided heuristics to limit speculation to coarse tasks in code regions where dependences are rare. T4 divides the entire program into tiny tasks, exposing speculative parallelism starting from the first instruction of `main()`. T4 uses the Fractal extension to Swarm [227] to compose parallelized code and perform speculative execution at the granularity of tiny tasks.

Figure 3-5 gives an overview of T4’s main components. T4 adds transformation passes within the LLVM/Clang compiler. All of T4’s passes are *intraprocedural*, that is, the compiler performs them on one function at a time, without relying on expensive *interprocedural* analyses. Thus, T4 compilation times stay small and proportional to code size.

T4’s passes run towards the end of the LLVM middle-end, after register promotion with SSA renaming, function inlining, and optimizations that reduce redundant or dead computation. Thus, T4’s passes operate on already simplified IR. This requires more engineering effort than implementing parallelization in a higher-level language representation. The reason we undertook this effort is that it significantly improves code generation by not impeding these standard compiler optimizations [206]. As we will see, this lets us produce parallel code that performs very little additional work compared to optimized serial code.

T4 parallelizes this sequential IR in four phases, which correspond to the following four sections of this chapter:

Section 3.4 – Reducing memory dependences: To improve parallelism and avoid false sharing, T4 first optimizes the allocation of local variables and avoids using a shared stack.

Section 3.5 – Decomposing programs into tiny tasks: T4 breaks *all* code into tiny tasks, using loop iterations and function calls as task boundaries. T4 tags tasks with timestamps to record program order, and produces spawners that can spawn many children tasks in parallel from within straight-line code.

Section 3.6 – Loop expansion: T4 generates spawner trees for loops, exploiting Swarm’s distributed task units for high-throughput task spawning. T4 can generate exponentially growing spawner trees even for a loop with unknown tripcount using *progressive expansion*.

Section 3.7 – Reducing communication costs: T4 *coarsens* tasks with striding accesses patterns and generates spatial hints to exploit locality. Finally, T4 *lifts* tasks into separate functions that can be spawned to hardware while reducing memory accesses by exploiting hardware support for register-to-register communication.

After this, the LLVM backend generates machine code.

3.4 Eliminating the Call Stack

Before parallelizing the code, T4 first transforms it to reduce false sharing caused by mingling of local variables on shared cache lines, and to eliminate false dependences caused by coordinating allocation and deallocation on a shared function call stack. To accomplish this, T4 uses novel compiler techniques: *bundling* local variables into heap allocations and selectively transforming functions with return values

into *continuation-passing style* (CPS). Together, these transformations maintaining nearly the same efficiency as a conventional stack while entirely eliminating the need to allocate function activation records on a shared stack.

Bundling: T4 replaces local variables that must normally be allocated on the stack by *bundling* them into a single heap chunk. A chunk is allocated at the start of its function and freed at the end. A variable is bundled only if the program explicitly uses pointers or references to it that prevent the compiler from promoting the variable to registers. Most local variables are not bundled, and are instead promoted to registers in the course of ordinary compiler optimization. When registers are full, T4 spills these values to task-private stack space [224] that is recycled at the end of each task.

Privatization: When bundling variables, T4 also *privatizes* local variables scoped within the body of a loop, allocating a separate instance for each iteration of the loop [154], just as is done by compilers implementing parallel programming frameworks such as OpenMP. This avoids false dependences between loop iterations.

CPS conversion: Continuation-passing style conversion eliminates the notion of a function returning to its caller. Thus, there is no need to allocate stack memory to form a record of function call frames to save stack frame pointers or return addresses.

T4 converts only function calls where the return value is used by the continuation to CPS. This conversion modifies the callee to accept an optional extra argument, a continuation *closure*. In our implementation, this closure is allocated on the heap and passed by reference. A closure's first field is a pointer to the continuation code, and subsequent fields hold values captured by the caller and used by the continuation. We also modify the caller to construct a continuation closure and pass it to the callee. At the end of the callee's execution, program control jumps to the continuation, with the return value passed in a register. For the code in Figure 3-2(a), this allows many calls to `f()` to be launched in parallel, without contention to allocate stack frames. CPS conversion also allows each call of `f()` to pass its return value `x` to a continuation without contending on a shared stack.

After the transformations of bundling, privatization, and CPS conversion, the sequential code is ready for parallelization. Many sources of false dependences have been removed and the code no longer uses a

shared stack, making it easier to spawn many tasks in parallel. Mutable local variables shared among tasks and immutable continuations passed to tasks with return values are now dynamically allocated on the heap. It is therefore critical that heap allocation (i.e. `malloc()`) be efficient, which we will address later in Section 3.8.

Comparison with prior work: Cactus stacks as used in fork-join parallel languages [67, 77] are the closest technique to T4’s stack elimination. Both techniques allow parallel tasks to dynamically spawn new tasks, including function calls, without contention for stack allocation. However, T4’s stack elimination technique has two important benefits over cactus stacks. First, T4 is selective in making heap allocations: whereas cactus stacks require a new heap allocation on each task spawn, most of T4’s tiny tasks do not require individual memory allocations. Second, by placing shared data in the heap, T4 separates shared variables from more frequently used local values on thread-private stacks. This avoids spurious conflicts that would occur if both types of data were mingled on a shared stack frame.

CPS conversion is well-studied in compilers for functional languages [12, 13, 222], where it is used as an intermediate representation that simplifies optimizations and eases code generation. We adapt CPS not to ease compiler implementation but to eliminate contention on the stack by replacing some stack allocations with heap allocations during speculative parallelization. To the best of our knowledge, we are the first to automate CPS conversion for sequential C and C++.

3.5 Full-Program Fine-Grain Parallelization

T4 exploits the structure of loops and functions to preserve program order while parallelizing an entire program with arbitrarily nested task spawns.

3.5.1 Task delineation

T4 divides code into tasks by turning each loop iteration, loop continuation, function call, and function call continuation into a separate task.¹

¹ While the rest of this dissertation primarily uses “task” in the sense of a *dynamic* task that is spawned at runtime, in this section we speak of dividing a program into *static* tasks which are the fragments of code that may be executed by dynamic tasks.

Like prior work, we find that this approach naturally limits tasks sizes to be fairly small [142, 248]. Programs do not commonly run thousands of instructions without encountering a function call or loop backedge.

Consider again the pattern of execution demonstrated by Figure 3-2(a): each loop iteration first does some computation, then performs a read-modify-write on some value in memory. That read-modify-write may incur conflicts and aborts. In the example in Figure 3-2(a), T4 automatically isolates the read-modify-write into a tiny task because it is a function call continuation.

3.5.2 Task-splitting annotations

While automatic task delineation suffices for compiling most code, tinier tasks are required to make aborts cheap under high contention. Thus, T4 also lets programmers annotate code regions to be split into tasks. Manual annotations help to isolate contentious memory accesses, which then enjoy the benefits of cheap aborts and spatial hints (Section 3.2.2). We find these annotations are not needed in code paths that execute infrequently and access rarely-updated data, so few annotations are needed to deliver large performance improvements even in large programs (Section 3.10.1).

These annotations are very easy to use because they do not affect program semantics: unlike the pragmas in conventional parallelization frameworks like OpenMP, it is impossible for T4’s annotations to be incorrect in the sense of changing the program’s output. As a result, selecting these annotations does not require careful thought: in our experience it is easy for a programmer to try several potential annotations within the hottest loops of an application and keep what performs well. We observe it improves the programmer experience to make the selection of what tasks to parallelize visible and controllable. For programmers who do not wish to make these selections themselves, we have demonstrated using profiling to identify the contentious memory accesses that cause aborts to allow automatic placement of fine-grain task boundaries [167].

3.5.3 Task ordering

T4 orders tasks with a combination of timestamps and Fractal domains. This process starts with the *control-flow graph* (CFG) of a function, a

length strings of timestamps are called virtual times, and they may be compared lexicographically to determine ordering between any two tasks (Section 2.3). A virtualization technique called zooming [227] allows hardware to maintain the semantics of unbounded-length strings while storing virtual times in fixed-width task descriptors. As an optimization that allows more timestamps to fit within fixed storage so zooming is needed less often, T4 flags most domains it creates where it has ensured tasks have unique timestamps, so hardware does not need to extend each timestamp with a tiebreaker to produce unique domain virtual times (Section 2.3).

3.5.4 Parallel task spawning

After dividing the sequential code into tasks, T4 determines where to place the spawn point for each task. T4 uses a heuristic that aggressively favors exposing parallelism and enabling selective aborts: task T spawns task U as a descendant only if T produces *live-out register values* used by U . For example, in Figure 3-2(a), since `f` produces a return value `x` used in the continuation, there is no point in spawning the call to `f` and its continuation in parallel. So T4 passes the continuation to `f`, which spawns the continuation when it has computed `x`. These separate tasks keep aborts cheap (Section 3.2.2). Otherwise, for example when considering a function call and continuation tasks that do not use any return value of the function, the tasks are spawned as siblings, allowing these tasks to run in parallel and be aborted selectively if needed (Section 3.2.1).

3.5.5 Simple task cleanup

After initial task boundaries, timestamps, and spawn points are decided, we encode all of this information the IR using an augmented version of the LLVM IR loosely based on Tapir [206]. The task structure can then be subsequently modified by further simplification or other optimizations.

Placing task boundaries using the heuristics in Section 3.5.1 may create needlessly small tasks, e.g., tasks that perform a small amount of computation only on registers and do not access memory. To eliminate these trivial tasks, we employ a simple cleanup pass that moves any code in tasks that do not access memory into the previous task and

erases empty tasks.

3.6 Parallel Loop Expansion

T4 adopts a multifaceted strategy to parallelize loops. Central to this strategy is the use of *spawners* as described in Section 3.2. T4 uses three compiler transformation strategies to *expand* loops to generate spawners: progressive tree expansion, bounded tree expansion, and chain expansion. Progressive expansion, a new strategy unique to T4, is the most critical one to parallelize programs with irregular control flow.

T4’s parallel tree loop expansion differs from all prior TLS compilers, which spawn iterations of any single loop serially. Exponentially expanding spawner trees expose asymptotically more parallelism: the critical path of task spawns grows logarithmically in the number of iterations, instead of linearly, and most task spawns are off the critical path. When T4 generates spawners, it also uses LLVM’s scalar evolution analysis to identify and eliminate induction variables that cause unnecessary dependences between iterations [246]. To avoid flooding the machine with tasks, each spawner is timestamped according to the first loop iteration it will spawn. This prioritizes spawners properly so the tree expands gracefully and tasks will not be spawned too far before they need to execute.

3.6.1 Progressive tree expansion

Progressive expansion generates spawner trees for loops where the number of iterations is unknown. Figure 3-7 shows progressive expansion in action on such an *unknown-tripcount* loop. Spawner tasks are shown in **gray** and loop iteration tasks are shown in **blue**. The number shown in each task is its timestamp. Figure 3-7 shows spawners that each spawn two loop iterations directly, then spawn two child spawners with doubled stride. Each spawner ensures its subtrees are balanced by giving its children spawners *interleaved* iterations. For example, in Figure 3-7, the spawner children of spawners 2 and 4 are *interleaved* (with timestamps 6 and 10, and 8 and 12). This exploits Swarm’s ability to spawn tasks in any order. T4 initiates loop execution by spawning the initial spawner task, `spawner(0, 2)`, which will lead to the eventual spawning of all loop iterations. This illustration shows a task tree with a fanout of 2 for simplicity. Our implementation varies the fanout of the spawner

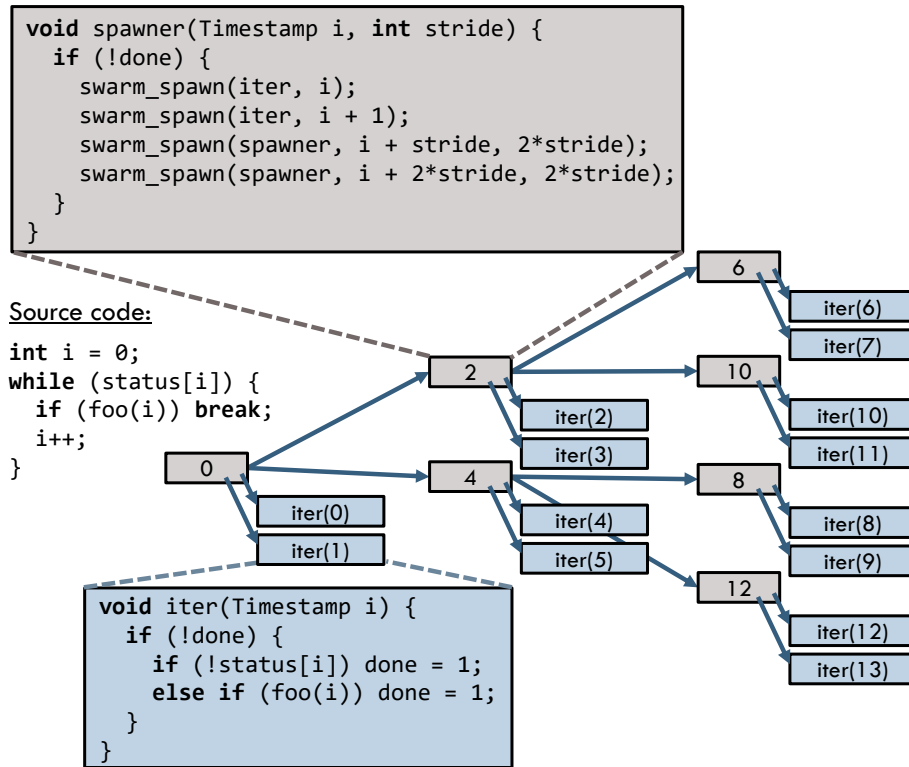


Figure 3-7: Progressive expansion scales a **while** loop. Boxes show pseudocode for tasks of the transformed loop. The call to **foo()** may spawn nested tasks (not shown).

tree to balance the demands of filling the system quickly and limiting the number of unneeded tasks spawned, beginning with each spawner spawning four loop iterations and four child spawners at early levels of the tree.

To handle unknown termination conditions, progressive expansion transforms control dependences into data dependences on a newly created termination flag variable. Control-flow paths that would exit the loop write to this flag, which is named **done** in the lower left of Figure 3-7. Loop iteration and spawner tasks read this flag and exit early if the loop has already terminated. The new flag is a memory-resident variable that exploits Swarm’s ordinary mechanisms for scalable speculation. Each task that reads the variable brings a shared copy into an L1 cache. When the flag is finally written upon loop termination, cache invalidations trigger the discovery of conflicts and abort any tasks that misspeculatively ran past the end of the loop.

Source code:

```
Node* ptr = start;
while (ptr) {
    ptr = ptr->next();
    do_body(ptr);
}
```

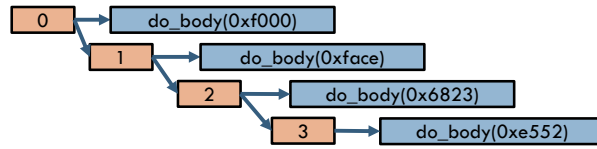


Figure 3-8: Chain expansion avoids conflicts on `ptr`. Executions of `do_body()` may spawn nested tasks in parallel (not shown).

3.6.2 Bounded tree expansion

In some loops, LLVM analyses can find an expression for the number of iterations. For example, in Figure 3-2(a), the compiler knows the tripcount will be given at runtime by the variable `size`. For these *known-tripcount* loops, T4 can generate spawner trees without control speculation: each spawner is responsible for a range of consecutive iterations, which it divides evenly across children spawners, forming a simple balanced tree as shown in Figure 3-4. Our implementation of this compiler optimization for known-tripcount loops is similar to that in Tapir [206]. However, to improve parallelism, we increase the fanout so each internal spawner spawns four children spawners, and each leaf spawner spawns up to eight loop iterations.

3.6.3 Chain expansion

While progressive expansion can operate on any loop, spawner trees are unprofitable for the loop in Figure 3-8, where each iteration depends on the previous iteration's update of `ptr`. T4 identifies such *serializing* variables meeting three conditions:

1. every iteration unconditionally reads and writes to it,
2. substantially all computation in the loop body is dependent on the value of the variable, and
3. the variable is not an induction variable that can be rewritten/eliminated.

In the rare condition all three conditions are met, T4 performs chain expansion which avoids aborts caused by speculating on these frequently updated variables.

As shown in Figure 3-8, chain expansion divides each loop iteration into two parts: a slice that computes the serializing value, shown in orange, and a slice that consumes it, shown in blue. Chain expansion performs well in outer loops where the consuming slice is large, con-

taining inner loops or function calls that T4 breaks into nested parallel tasks. As an optimization, T4 can schedule **orange** tasks to run at one location so spawn rate is insensitive to communication latency, as in SpecDSWP [244].

3.7 Reducing Communication Costs

T4 performs optimizations that reduce data movement costs among tasks: exploiting locality by coarsening tasks with striding accesses and generating spatial hints, and packing live-in values to reduce task-spawn costs.

3.7.1 Loop task coarsening for cache alignment

To avoid aborts when hardware falsely detects dependences on accesses to different addresses within a shared cache line, T4 identifies inner loops that scan through memory with a fixed stride per iteration and *coarsens* the tasks associated with the loop to make tasks cache-aligned: each coarsened task covers all the elements of one or more cache lines, and consecutive tasks access disjoint cache lines. For example, with 64-byte cache lines and accesses that stride 48 bytes per iteration, T4 coarsens by a factor of four and generates prolog and epilog code so that each task covers its own three cache lines. We adapt strip-mining and prolog loop generation from SIMD vectorization [29, 131]. However, while automatic vectorization relies on fragile pointer analyses or programmer annotations (e.g., the **restrict** keyword) to prove memory accesses are safe to parallelize [146], T4 coarsens loop tasks without relying on aliasing guarantees, eliminating false sharing due to striding accesses in many inner loops.

Salamanca et al. had previously observed strip-mining could avoid false sharing for loops with striding access patterns [201]. T4’s loop task coarsening fully automates this strip-mining, including strip size selection, and generates prolog and epilog loops to align task boundaries to cache lines, instead of assuming array accesses always start at the beginning of a cache line.

3.7.2 Spatial-hint generation for locality-aware speculation

Spatial hints exploit spatial locality even for irregular access patterns. To generate spatial hints, T4 identifies tasks that write to a single memory address, or a related set of memory addresses (e.g., addresses falling within a single cache line), as candidates to be given spatial hints. For each candidate task, T4 checks if it can hoist the computation of the accessed address into the parent task. This hoisting is easy for tiny tasks, where the nearest task boundary preceding a memory access is close by. If the address computation is successfully hoisted, then T4 uses the *cache line address* (computed by a right shift) as the hint. This reduces ping-ponging of frequently written cache lines (Section 3.2.2). Section 3.9 presents a case study where this is crucial to obtain scalability.

3.7.3 Task lifting with live-in reduction

Up to now, T4 represented task code nested within larger functions, using an approach similar to Tapir [206]. This allowed task transformations to reuse existing LLVM analyses on control- and data-flow across task boundaries. To finalize task boundaries, T4 *lifts* each task’s code into a separate LLVM IR function, which requires each spawn site to capture the task’s live-in values in a closure.⁴ If a task’s live-ins fit into the registers that hardware task descriptors can hold (five 64-bit values in addition to the function pointer and timestamp in our implementation), the entire closure is passed through registers, avoiding memory accesses. If live-ins do not fit in registers, the parent task allocates the extra live-ins on the heap and passes a pointer so the child task can read the values.

T4 performs three optimizations to reduce closure sizes:

1. *Loop environment sharing* allocates a single heap object that holds all loop-invariant live-ins. All loop iterations read from this single location, achieving good L1 hit rates.
2. *Live-in sinking* finds task live-ins that can be very cheaply recomputed from other available values (e.g., multiple addresses computed by adding constant offsets to the same pointer) and sinks or copies the computation into the task [6].
3. *Register packing* generates instructions to shift remaining live-ins

⁴ Much like lambda lifting or closure conversion [12, Chapter 10].

into the minimum number of machine registers. For example, real-world programs often use 32-bit integers, and T4 can pack two 32-bit live-ins into a 64-bit value for task spawns. T4 adds instructions at the start of the task to unpack the live-ins into separate registers. These optimizations trade increased instruction count for reduced task-spawn data movement. With these optimizations, most task spawns need only capture a few live-in registers into a small task descriptor, which is cheap to send over the on-chip network and hold in hardware task queues.

3.8 Scalable Speculative Memory Allocation

The scalability of dynamic memory allocation (i.e., `malloc()`) is critical to the performance of general parallel programs [25]. Not only do some applications make intensive use of heap allocation, T4’s transformations create additional heap allocations for local variables shared among tasks. We therefore need a suitable memory allocator for T4 that allows speculative parallel tasks to perform allocation and deallocation without gratuitous serialization.

Unfortunately, the semantics of dynamic memory allocation makes it inefficient or inappropriate to call into existing allocators within hardware-managed speculative tasks. One could use an allocator not designed for speculative execution, but hardware would serialize tasks that read and update the allocator’s internal metadata (i.e., its freelists). Most of this serialization is not material to program semantics. For example, it is safe for parallel tasks to allocate any distinct available chunks out of order. Furthermore, if a task allocates memory and then aborts, it is safe to simply deallocate the memory: it is not necessary to further abort parallel tasks that had allocated from the same free lists. Prior work has developed specialized allocators for transactional-memory systems without speculative forwarding to avoid needlessly serializing accesses to allocator metadata [98], e.g., by using escape actions or open-nested transactions [272]. However, these techniques are unsafe in systems with speculative forwarding such as most prior TLS systems as well as Swarm. As far as we know, no scalable memory allocator has been implemented for systems with speculative forwarding.

capalloc: To solve this problem, we implement `capalloc`, a memory allocator built with Capsules (Section 2.3.4). In `capalloc`, all allocation

calls (`malloc()`, `calloc()`, etc.) are capsule functions, which register an abort handler that deallocates the memory if the calling task aborts. To avoid reusing memory freed by tasks that later abort, each deallocation call (`free()`, etc.) spawns a *commit handler*, a special type of non-speculative task [109], to perform delayed deallocation. After the parent task commits (or aborts) the TSB sends its children commit handlers (or abort handlers) to TQs which will dispatch them to run non-speculatively on the next available core. Unlike ordinary tasks, we do not assign these handlers timestamps for them to participate in the commit protocol, and hardware commits these tasks immediately when they run. This is critical for performance, as these handlers’ execution is delayed until they are safe to run non-speculatively, so their participation in Swarm’s commit protocol would delay commit of other ordinary tasks.

`capalloc`’s internal design mimics TCMalloc [73], a state-of-the-art and widely used multithreaded memory allocator. Small allocations (≤ 16 KB in our implementation) are served by a per-core software cache. Per-core caches are accessed without locks and retain recently used chunks at the same core, improving locality. 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. Unlike TCMalloc, which reuses free chunks to implements freelists as linked lists, `capalloc` keeps all its metadata in untracked memory (Section 2.3.4), so hardware does not observe data dependences on freelist updates. Thus, parallel speculative tasks can perform allocations without spurious serialization: a high-timestamp task that speculatively allocates memory will not be aborted if a lower-timestamp task later arrives and allocates from the same freelist.

3.9 Putting It All Together

We now present a case study on how T4’s techniques combine to parallelize a challenging loop from `473.astar`. Due to its frequent data dependences, no prior TLS system has reported significant speedup on `astar`.

Listing 3.1 shows a hot loop in `astar`, which finds paths in a 2D grid. Each loop iteration visits a node identified by `index`. For each of

```

1 int bound2l = 0;
2 for (i = 0; i < bound1l; i++) {
3     index = bound1p[i];
4     index1 = index-yoffset-1; // NW neighbor
5     // 1st of 8 identical code blocks:
6     if (waymap[index1].fillnum != fillnum
7         && maparp[index1] == 0) {
8         bound2p[bound2l++] = n;
9         waymap[index1].fillnum = fillnum;
10        waymap[index1].num = step;
11        if (index1 == endindex) {
12            flend = true;
13            return bound2l;
14        }
15    } // End of 1st identical code block.
16    index1 = index-yoffset; // N neighbor
17    // 2nd of 8 identical code blocks appears here.
18    index1 = index-yoffset+1; // NE neighbor
19    // 3rd of 8 identical code blocks appears here.
20    // ... more repetitions w/ different index1 ...
21    // 8th of 8 identical code blocks appears here.
92 }

```

Listing 3.1: Code taken from Way_.cpp in *astar*, compressed and with repeated code blocks omitted

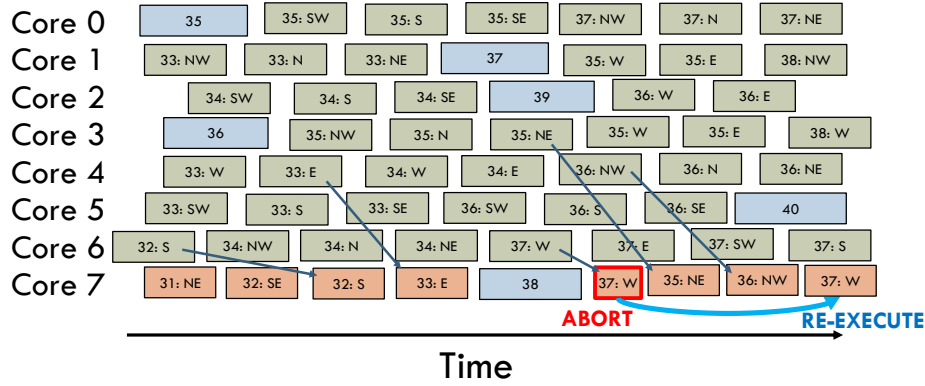


Figure 3-9: Execution timeline of *astar*. Task colors match Listing 3.1. Numbers denote loop iterations. Arrows indicate spawns of neighbor-append tasks.

the node's eight neighbors in the grid, identified by values of `index1`, the code checks whether the neighbor should be appended to a queue, `bound2p`, to be visited later. This happens about once per iteration on average.

However, to avoid frequent, expensive aborts, we must isolate

neighbor-appends, such as on line 7, as each append depends on the previous update to `bound21`. T4 delineates tasks as shown in Figure 3-9: each iteration starts with a **blue** task which spawns eight neighbor-checking tasks, in **green**, which have been split into tasks by annotations and use spatial hints (Section 3.7.2) to exploit locality in the `waymap` array. Neighbor-checking tasks that access different cache lines within `waymap` can run in parallel. Neighbor-checking tasks spawn neighbor-append tasks, in **orange**, when needed. Neighbor-append tasks use the cache line of `bound21` as a spatial hint, so accesses to this contentious variable execute serially in one tile. This is critical to achieve any speedup (Section 3.10.4). Aborts still occur if tasks execute out of order, but Swarm’s timestamp prioritization limits the frequency of aborts.

To keep cores busy, T4 must parallelize neighbor-checking tasks across loop iterations. This is an unknown-tripcount loop due to **return** statements inside the loop, so progressive expansion is needed to allow for spawning iterations quickly.

3.10 Evaluation

3.10.1 Experimental methodology

Simulated hardware: We use a cycle-level simulator based on Pin [143, 173] to model Swarm systems with parameters given in Table 3-2. We use detailed core, cache, network, and main memory models, and simulate all task and speculation overheads (e.g., task traffic, running misspeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). We also simulate smaller systems with fewer tiles, keeping per-core cache sizes and queue capacities constant. This is equivalent to using partitions of the 64-core system.

To reduce wasted work, we also implement a gentle hardware heuristic for throttling overly eager speculation. If a task has been repeatedly aborted, the task unit delays its next dispatch, and the delay grows with additional aborts. We find this gentle throttling does not hurt performance in any benchmark. In fact, it helps execution time for benchmarks with a high ratio of aborts, by avoiding cache-line ping-ponging and network contention created by misspeculating memory operations.

Benchmarks: We evaluate T4 with real-world applications from

Cores	64 cores in 16 tiles (4 cores/tile), 3.5 GHz, x86-64 ISA; Haswell-like 4-wide OoO superscalar [82]
L1 caches	32 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	1 MB, per-tile, 8-way, inclusive, 9-cycle latency
Last-level cache	64 MB, shared, static NUCA [121] (4 MB bank/tile), 16-way, inclusive, 12-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	Four 4×4 meshes, 192-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns
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)
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [45]
Commit	Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Spills	Tiles send updates to virtual time arbiter every 100 cycles
	Spill 15 tasks when task queue is 86% full

Table 3-2: Configuration of the 64-core system.

Benchmark	Lines of code	Modified lines	Cycles per task
429.mcf	1,574	None	236
433.milc	9,575	+18, -13	153
444.namd	3,887	None	1772
450.soplex	28,302	+25, -16	38
456.hmmer	20,680	+11, -9	170
462.libquantum	2,605	None	211
464.h264ref	36,032	+12, -9	25
470.lbm	904	+1, -1	809
473.astar	4,285	+29, -144	11
482.sphinx3	13,128	+17, -8	47
Total	120,972	+114, -201	347

Table 3-3: SPEC CPU2006 benchmarks used in the evaluation. Lines of code exclude comments and whitespace.

SPEC CPU2006 [89]. We exclude the ten Fortran benchmarks, which are floating-point, scientific applications for which manually parallelized versions already exist. The C/C++ programs we evaluate are hard to parallelize, even with speculation [59, 117]. To the best of our knowledge, Packirisamy et al. obtained the best previous speedups for these

benchmarks— $1.6\times$ at 4 cores, $1.91\times$ at 8 cores—by combining advanced compiler techniques with hardware support for TLS and synchronization [171].

Several benchmarks cannot be compiled by T4 because they use features whose existing implementations rely heavily on a conventional stack layout, such as C++ exceptions or `setjmp()/longjmp()`.⁵ We evaluate all ten benchmarks that can be compiled by T4, listed in Table 3-3.

As T4 is based on LLVM/Clang 5.0, we compare the results of compiling each benchmark with T4 v.s. an ordinary serial binary compiled with LLVM/Clang 5.0. Both versions are compiled with `-O3`. We verified that T4-compiled benchmarks deterministically produce the same result as the serial version.

We evaluate all benchmarks with the `ref` (largest) inputs. Since these run for a very long time, we use SimPoints [175, 210] to select a sample period of the serial version’s execution containing 2 billion dynamic instructions that is representative of steady-state execution. None of the benchmarks spends the sample period in a single loop: each sample includes many transitions between loops and other hot code regions.

To ensure we simulate the same sample region regardless of compiler transformations, we automatically instrument certain function entry points with *heartbeats*. Our simulator counts heartbeats to determine the start and end of the sample period.

We modify the source code of some of the benchmarks to help the compiler uncover more parallelism. Table 3-3 reports the lines changed, a small fraction of the program’s lines of code in all cases. In `soplex` and `sphinx3`, these modifications are annotations to break up tasks as explained in Section 3.5.2. In `astar`, we deduplicate repetitive code and add annotations to break up tasks as shown in Section 3.9. We manually perform loop fission in `hmmmer` to allow T4 to better divide different striding memory accesses into parallel tasks that access separate cache lines (Section 3.7.1). This loop fission can be automated in future work. `lbm`, `milc`, and `h264ref` use simple modifications to avoid false sharing. In `milc` and `h264ref`, we move the declarations of variables into loops when they are used to hold short-lived values within each loop iteration. This enables T4’s privatization to avoid false dependences

⁵ Future work could address this, e.g., by passing escape continuations [12, 223]. This needs non-trivial engineering due to how exceptions are handled in LLVM.

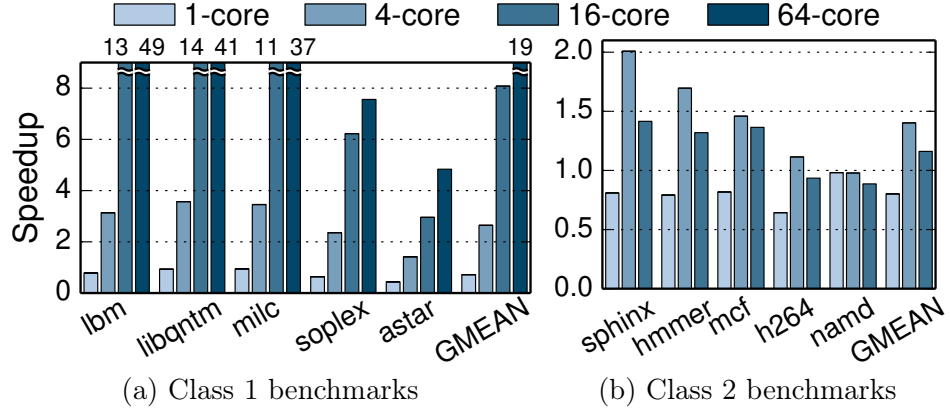


Figure 3-10: For each system size, performance of T4 normalized to serial code running on the same system.

(Section 3.4).

We organize the evaluation as follows: we analyze T4’s overall performance (Section 3.10.2), how T4’s spawner trees are key to performance (Section 3.10.3), the effect of task-splitting annotations and hints (Section 3.10.4), task lifting optimizations (Section 3.10.5), and sensitivity to core microarchitecture (Section 3.10.6).

3.10.2 T4 performance

Figure 3-10 reports the performance of T4 (higher is better) relative to serial code on systems with 1, 4, 16, and 64 cores. We divide benchmarks into two classes: Class 1’s hottest inner loops have some iterations that are independent of other iterations, while Class 2 benchmarks’ hottest loops contain mutable scalar variables accessed on *every* iteration. Such data dependences necessarily limit the scalability of Class 2 benchmarks, so we focus on Class 1 benchmarks. T4 scales Class 1 benchmarks well, achieving gmean $19\times$ speedup. T4 scales multiple benchmarks to tens of cores, with up to $49\times$ speedup on 64 cores (for **lbm**).

Figure 3-11 provides further insight into these results. Each bar’s height shows the execution time (lower is better) of T4 at 1, 4, 16, or 64 cores, relative to the execution time of the original serial version on 1 core. The 1-core T4 bars show that *T4 introduces modest overheads* of gmean 32% across all ten benchmarks. The majority of the overhead is from instructions T4 generates to pack and unpack live-ins when spawning tasks or to load values from shared loop environments (Section 3.7.3).

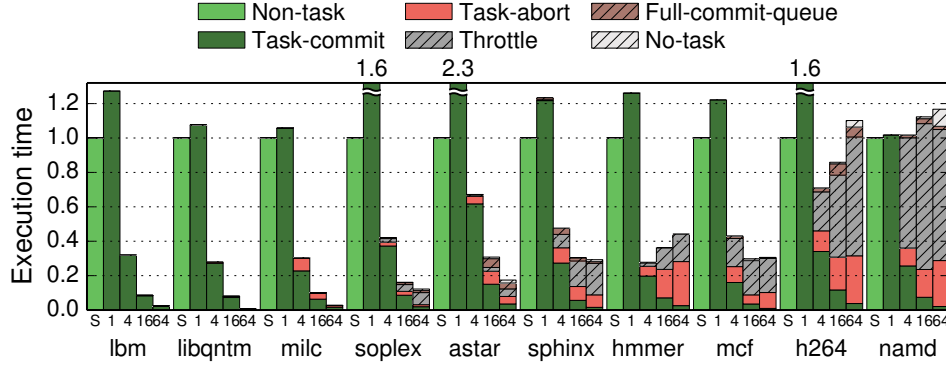


Figure 3-11: Breakdown of execution time of T4 systems with 1, 4, 16, and 64 cores, normalized to the Serial code on a 1-core system.

T4 has the highest overheads in *astar*, *soplex*, and *h264ref*, which are divided into very tiny tasks, tens of cycles long, as shown in Table 3-3. In return, short tasks make aborts cheap and confer high speedups.

Figure 3-11 also shows a breakdown of how cycles are spent, averaged across cores. The breakdown shows how cores spend time in five ways:

1. executing tasks that later commit,
2. executing tasks later abort,
3. idle because of the throttling heuristic,
4. stalling on a full commit queue, or
5. idle because no task is available to run.

This breakdown shows that, on all benchmarks at 4 cores, cores spent most of their time executing useful work that will commit. Because Class 2 benchmarks do not scale beyond 4 cores, their execution time changes little at higher core counts, with the additional cores mainly running more speculative tasks that abort or are throttled when tasks abort repeatedly. Class 1 benchmarks scale to tens of cores, and aborts take a minority of execution time even at high core counts. Cores rarely stall on task commit, as commit queue (Section 2.3) occupancy averages 43 buffered tasks waiting to commit per tile, across all benchmarks. Cores also rarely lack tasks to run, as task queues average 24 runnable tasks per tile. Timestamp-prioritized spawners (Section 3.6) unfold trees gradually, so time lost to spilling task queues is $<0.01\%$.

Class 1 benchmarks: *lbm*, *libquantum*, and *milc* have plentiful parallelism. Prior work has noted that data dependences between iterations of their inner loops are rare, but achieved limited scalability due to inefficient mechanisms for spawning large numbers of speculative

tasks [171]. T4 scales these benchmarks with highly parallel spawner trees.

T4 also finds significant parallelism in `soplex` and `astar`, yielding $7.6\times$ and $4.8\times$ speedups, respectively. Data dependences in these benchmarks do cause noticeable aborts, but because these aborts are selective, they do not impede the scalability of independent work. The majority of `soplex`'s hot inner loops resemble the example in Figure 3-2(a). Meanwhile, `astar`'s inner loops are described in Section 3.9. Both have read-modify-write operations and feature significant spatial locality that spatial hints exploit.

Class 2 benchmarks: These benchmarks are dominated by loops where each iteration unconditionally accesses a shared mutable variable, creating a long critical path of data dependences that limits parallelism. In `sphinx3` and `mcf`, T4 extracts parallelism by isolating dependences into tiny tasks. In `sphinx3`, the tiny tasks are generated due to manual annotation (Section 3.10.4). Meanwhile, `mcf` spends much of its time in pointer-chasing loops in which the data dependence is at the top of the loop body. T4 pipelines the execution of loop iterations using chain expansion (Section 3.6.3), without requiring any source code annotations, with performance similar to prior TLS systems.

Parallelism is limited in `hmmer` and `h264ref`, and T4 yields no speedup on `namd`. In these benchmarks, inner loops have many true loop-carried dependences and little independent work that can run in parallel. Prior work obtains similar or slightly higher speedups on Class 2 benchmarks [171] through aggressive compiler optimizations that move instructions off the critical path [263]. These compiler techniques could be implemented in T4 to improve the performance of these benchmarks. However, T4 focuses on applications where spawner trees, tiny tasks, and spatial hints can avoid a rigid critical path, allowing tasks to run in parallel without every loop iteration participating in a serial bottleneck.

3.10.3 Benefits of spawners and spawner trees

Benchmarks show large variability in scalability, but T4's uses of spawners yield broad benefits across many benchmarks.

Figure 3-12 compares the execution time of three T4 variants with an idealized TLS baseline. All experiments use small tasks, spatial hints, and the same Swarm hardware support for task spawns, task commit, and conflict detection. Figure 3-12(a) shows 64-core results for

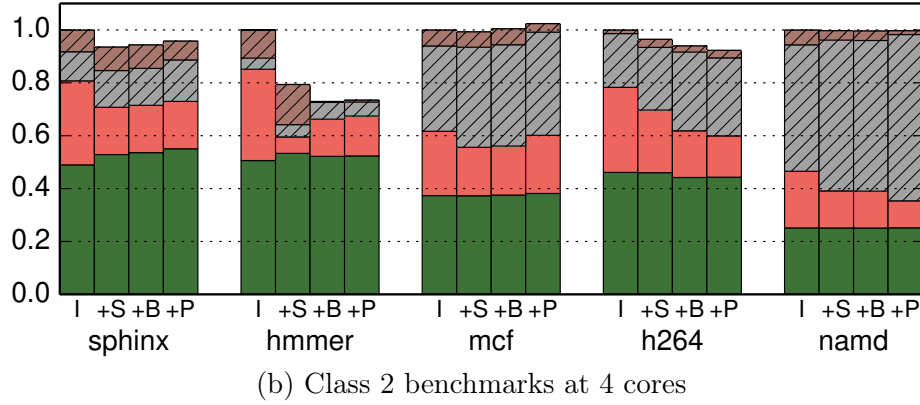
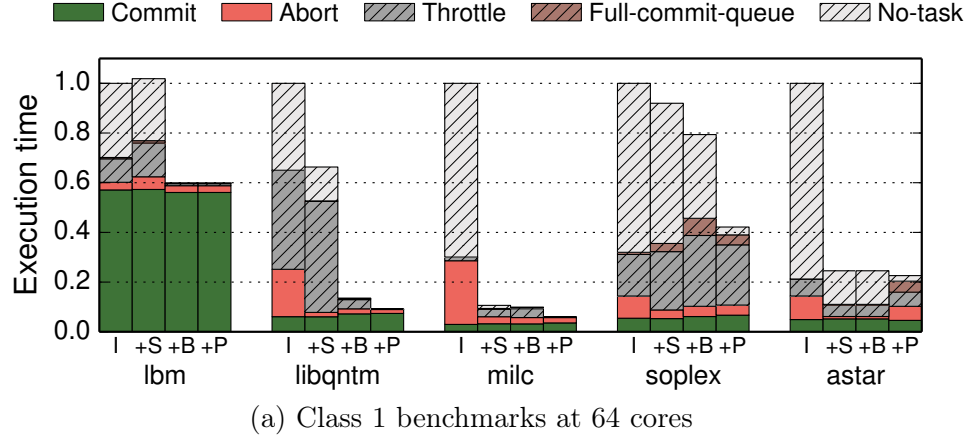


Figure 3-12: Execution time for Ideal TLS baseline, and T4 when successively enabling more features: Selective aborts, Bounded tree expansion, and finally Progressive expansion.

Class 1 benchmarks (as in Figure 3-1). Figure 3-12(b) shows Class 2 benchmarks at four cores, since their performance does not meaningfully improve beyond four cores.

In the TLS baseline, function calls and nested loops have the benefit of parallel spawns, but iterations within each loop are spawned one at a time, as in prior work [142, 194]. When a conflict is detected, it is broadcast to all tiles, which perform an *idealized abort and rollback of later tasks* in a single cycle. We idealize aborts to show that TLS has limitations that go beyond hardware bottlenecks.

The first T4 variant (+S) uses spawners to enable Swarm’s selective aborts (Section 3.2.1). Enabling selective aborts not only leads to less wasted work, it also reduces no-task cycles, because in the moments

following an abort, more independent tasks are still available to run. This benefits all benchmarks but `lbm`, which had few aborts to begin with. `lbm +S` is slightly worse than the ideal baseline because we are now modeling the latency of rolling back writes on each abort.

The middle T4 variant (+B) enables bounded spawner trees for known-tripcount loops (Section 3.6.2). This brings significant benefits to the hottest inner loops in `libquantum`, `lbm`, and `soplex`. The final T4 variant (+P) enables progressive expansion to scale unknown-tripcount loops, which appear in every benchmark except `lbm`. By spawning work more quickly, spawner trees reduce the time that cores spend idle. Progressive expansion also has a mixed effect on aborts: running more speculative tasks may result in more aborts, but populating tasks queues more quickly can also reduce aborts in some benchmarks, as task queues can better prioritize the dispatch of less-speculative (lower-timestamp) tasks.

3.10.4 Task-splitting annotations and spatial hints

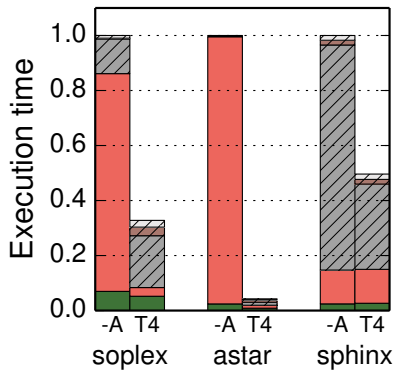


Figure 3-13: 64-core execution time without (-A) and with (T4) task-splitting annotations.

Figure 3-13 shows the performance impact of disabling code annotations that instruct T4 to split tasks at finer granularity (Section 3.5.2) for the three benchmarks that use annotations: `soplex`, `astar`, and `sphinx3`. These benchmarks are dominated by loops where most computation is spent on parallelizable work to compute updates to apply to shared data, and little time is spent updating the shared data itself. Without annotations, the only task boundaries come from loop iterations and function calls, leaving the expensive parallelizable computation in the same task as the conflict-prone memory accesses. Thus, the three benchmarks become dominated by expensive aborts and throttling

of aborting tasks. (If throttling is disabled, total execution time is even worse.) This shows the importance of identifying accesses to contentious data and isolating them with tiny tasks, which makes aborts cheap and enables spatial hints (Section 3.2.2). Because task-splitting annotations

cannot change program output, it is safe to try inserting them in different positions without careful analysis and then keep what works best. As shown by Table 3-3, we annotated a very small portion of the code in these three benchmarks.

T4 is the first compiler to obtain any meaningful speedup on **astar** or **soplex**. Prior TLS efforts could not even achieve $2\times$ speedup for **astar** on any number of cores. This shows the importance of using tiny tasks that enable spatial hints (Section 3.7.2). Hints are critical to scalability, as otherwise a single frequently written cache line can cause copious aborts.

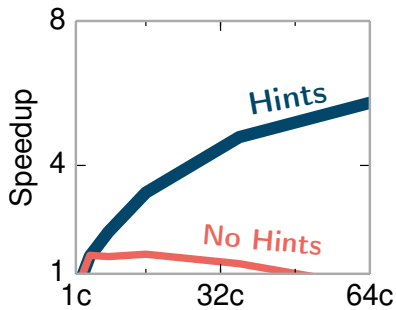


Figure 3-14: **astar needs spatial-hint generation.**

When **astar** repeatedly updates a single variable **bound21**, these serialized operations form a critical path that limits whole-program performance (Section 3.9). Figure 3-14 shows that spatial-hint generation is critical to scale **astar**, and without it T4 would fall flat, achieving $1.5\times$ speedup on 16 cores and deteriorating thereafter. Without spatial hints, tasks are sent to random tiles, ping-ponging the cache line in which **bound21** resides.

3.10.5 Task spawn traffic and effect of live-in reduction

T4’s optimizations produce tasks with few live-ins (i.e., inputs). This is crucial, because it enables spawning tiny tasks and distributing them across a large chip with small bandwidth costs. Table 3-4 shows the volume of live-in values transferred from parents to children tasks, with and without loop environment sharing and live-in sinking (Section 3.7.3). These optimizations reduce the number of register values that must be saved into task descriptors and shipped across the network, which are the main overheads associated with task spawning. With T4’s optimizations, spawning most tasks only requires moving a few 64-bit register values (Section 2.3), much cheaper than prior TLS systems that moved whole register contexts. Table 3-4 shows that most of T4’s task spawns do not allocate memory.

Benchmark	No optimizations		With optimizations	
	Bytes	Tasks	Bytes	Tasks
	per task	using heap	per task	using heap
lbm	21.6	0%	18.8	0%
libqntm	26.9	17%	12.0	0%
milc	21.5	1%	20.4	0%
soplex	29.8	14%	17.0	0%
astar	63.0	67%	24.7	0%
sphinx3	64.7	26%	26.8	4%
hmmer	90.2	86%	26.6	18%
mcf	148.7	97%	28.6	3%
h264ref	54.0	66%	22.4	5%
namd	175.1	34%	185.8	33%
Average	69.6	41%	38.3	6%

Table 3-4: T4’s task lifting optimizations reduce NoC traffic due to task live-ins and reduce heap allocations needed to pass data between tasks.

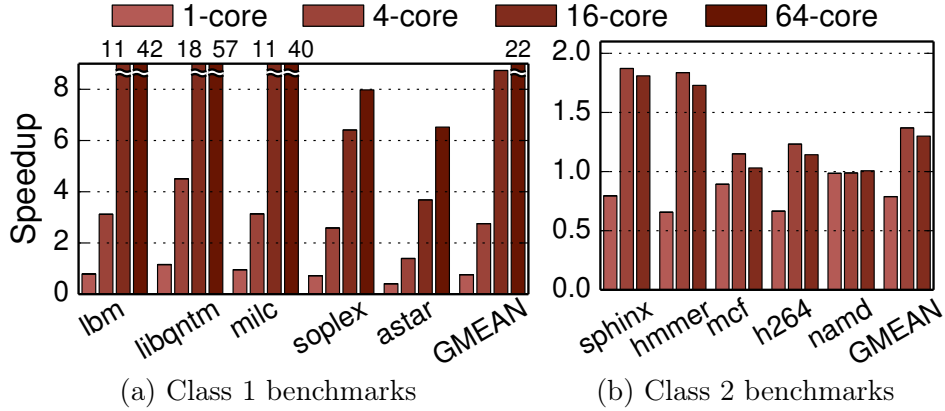


Figure 3-15: Performance of T4 on systems with simple in-order cores, normalized to serial code running on the same system.

3.10.6 Core microarchitecture

Results so far use up to 64 superscalar out-of-order (OoO) cores, reflecting real products available in 2020. But T4 is orthogonal to core microarchitecture. For comparison, Figure 3-15 shows the performance of T4 on systems with scalar, in-order cores. Despite the very different core microarchitectures, simple in-order cores show T4 enjoys similar speedups as with OoO cores in Figure 3-10. OoO cores run gmean $2.6\times$ faster than simple in-order cores by exploiting ILP within each task,

while T4 exploits speculative parallelism over hundreds of tasks, a far larger window than OoO cores.

3.11 Additional Related Work

T4 shares a similar goal to prior TLS systems [63], many of which were previously discussed in Section 2.2 and Section 3.1. These prior TLS systems were limited by the hardware they proposed to use and could not scale to tens of cores as T4 does. For example, because they lacked access to Swarm’s hardware support for selective aborts and spatial hints, prior TLS compilers did not have the opportunity for optimizations similar to T4’s for reducing abort costs and providing locality-aware execution. In some cases, hardware sufficient to implement some T4 optimizations was proposed prior to Swarm, which we discuss below.

3.11.1 TLS with out-of-order spawn

T4’s separation of spawners from worker tasks is not possible in most TLS architectures, where a task cannot spawn multiple children that commit. Renau et al.’s TLS architecture [194] relaxes the requirement that task spawns must be serial, allowing speculative tasks to spawn children independently. This architecture can exploit some nested parallelism to form some task trees, and in fact it enabled the POSH compiler to perform whole-program speculative parallelization [142]. But it still performed serial commits that limited performance. They addressed serial per-task commit overheads by dynamically merging tasks when cores are all occupied, resulting in committing fewer, coarser tasks at the cost of more expensive aborts.

Moreover, rather than allowing the compiler to specify a commit order independent of spawn order as Swarm does with timestamps, Renau et al.’s architecture requires that any single task must spawn its children strictly in reverse of the order they should commit. This prevents general forms of nested parallelism that require interleaved ordering. They could not interleave task timestamps as needed for progressive expansion. In fact, Renau et al.’s architecture and the POSH compiler that targets it still spawn iterations serially when parallelizing a single loop.

Finally, Renau et al. did not decouple spawn order from execution

order by prioritizing dispatch of less-speculative (i.e., lower-timestamp) tasks as Swarm does. Instead, cores immediately execute a task when it is spawned, even if the task is very speculative and unlikely to commit. By contrast, Swarm’s distributed task queues manage many tiny tasks and prioritize allocating resources such as cores to tasks that will commit soon. This makes it profitable to spawn tasks before they need to execute so the task queues will be able to prioritize tasks before they dispatch to a core. Thus, while POSH had to use profiling to avoid spawning tasks that were likely to abort, T4’s design can instead spawn tasks aggressively to unlock greater parallelism.

3.11.2 TLS with register-to-register communication

Many TLS compilers communicate values through memory, e.g., by preventing the promotion of shared values to registers so that they are always resident in memory at task boundaries [142]. This impedes standard compiler optimizations [193], and the energy and bandwidth requirements of moving entire cache lines for every task spawn presents a challenge to scaling tiny tasks to many cores. Some TLS systems introduce hardware mechanisms to send register values among tasks and allow software to select which registers to send [36, 125, 218]. T4’s register packing and task-lifting optimizations would reduce the volume of data sent in such architectures, making task spawns cheaper for them just as it does for Swarm.

3.12 Summary

We have presented T4, a compiler that takes a new approach to speculatively parallelize sequential programs. T4 uses task trees to spawn tasks far in advance, exploiting Swarm hardware to fill the system with work. Trees make tiny tasks practical. In turn, tiny tasks unlock new opportunities to make aborts cheap and expose more parallelism. To efficiently spawn tiny tasks, T4 introduces novel transformations such as progressive expansion to produce exponential trees for irregular loops, and a reimagined form of CPS conversion to eliminate contention on the stack. T4’s task-lifting optimizations make task spawns cheap, and tiny tasks enable spatial-hint generation, which makes T4 the first system for speculative parallelization to use dynamic information from sequential code to exploit locality.

By combining new software and hardware techniques, T4 broadly improves scalability over prior work, extracting enough parallelism to keep tens of cores busy, including on irregular code that defeated prior work.

4 S5: Schedule Structures to Sidestep Spurious Serialization

Part of this work was conducted in collaboration with Ajay Brahmashatriya, Emily Furst, Claire Hsu, Changwan Hong, Yunming Zhang, Julian Shun, Daniel Sanchez, and Saman Amarasinghe. Specifically, the GraphIt DSL compiler used for some workloads in this chapter as developed collaboratively. S5's design and implementation are individual contributions of this dissertation.

4.1 Introduction

The usage patterns of data structures greatly affect the parallelizability of a program. Parallel tasks may read shared data without serialization, so it is data that receive frequent writes that are of concern. Fortunately, if a large data structure's constituent elements have many writes as their state evolves over time, writes to separate elements may be executed in separate parallel tasks. However, there may be limited parallelism in the program phases when a data structure is first populated or when it is finally emptied. We observe parallelism in real-world sequential programs is commonly bottlenecked by data structures that have only these two phases. That is, after these structure's elements are produced, the program only accesses the elements to consume them. We call such data structures *schedule structures* because they schedule deferred consumption of data, or in other words, *they act as task queues*.

For example, Figure 4-1(a) shows a program using `std::vector`, the commonly used C++ datatype for a dynamic array. The vector named `valuable_widgets` is a schedule structure that collects pointers in one

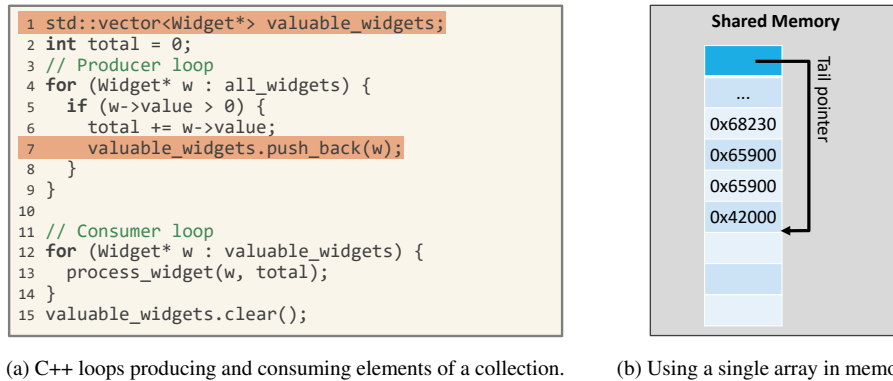


Figure 4-1: C++ code using `std::vector` to facilitate deferred work.

loop in order to consume them in another loop later in the program. This vector is efficient for serial execution because it is simple and compact, but vector insertions cause data dependences that serialize execution. This serialization is gratuitous. `valuable_widgets` merely serves to defer the task of passing a pointer to `process_widget()`, which could be done without storing elements in a vector.

The gratuitous serialization arises from within the implementation of `std::vector`, which has a pointer that tracks where to write the next element as shown in Figure 4-1(b). Calling `std::vector::push_back()` increments this pointer to the next contiguous location in memory. Thus, different cores performing insertions would ping-pong this pointer’s cache line from one core to the next, serializing all insertions. Unfortunately, this type of serialization occurs not only when appending to vectors, but when populating many schedule structures used in sequential programs. The typical usage of arrays, linked lists, trees, and hash tables in sequential programs nearly always updates some pointer, index, or count upon every insertion, causing copious serialization.

To address this problem, this chapter presents S5, a system that transforms schedule structures to be nearly as fast as a scheduler implemented in fixed-function hardware while also being as flexible as software schedulers that can be customized for an application’s specific needs. The name S5 describes how it enables efficient implementations of schedule structures to sidestep spurious serialization. S5 combines three components: a library of simple *abstract data types* (ADTs) that capture an application’s scheduling needs, a new compiler named T4+ that extends T4 to transform scheduling into parallel task spawns, and a new architecture that extends Swarm to let software customize the

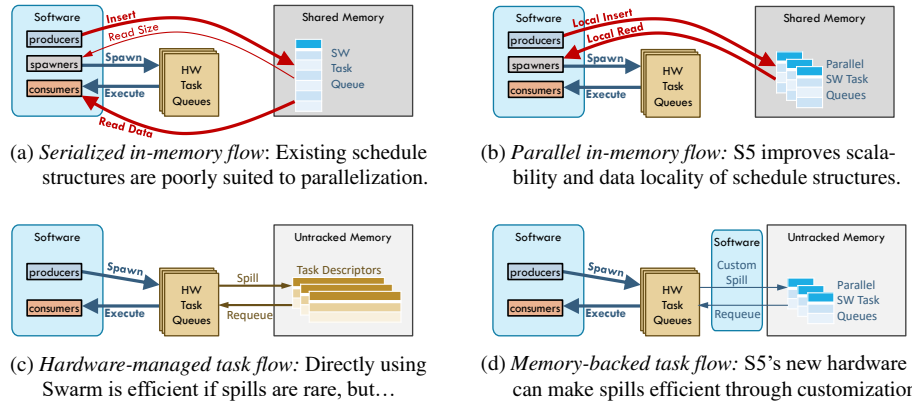


Figure 4-2: Four flows for tasks producing elements of a collection to launch tasks that consume the elements. Hardware queues are fast and placed near cores, while software queues are customizable and have larger capacities.

semantics of hardware-managed tasks.

Figure 4-2 shows four ways consumer tasks can flow into Swarm’s hardware queues that manage implicitly parallel execution. T4 can only exploit limited parallelism in a schedule structure such as that in Figure 4-1 by using the *serialized in-memory flow* in Figure 4-2(a). This is because, while T4 can optimize the memory layout of local variables, T4 cannot automatically transform dynamic data structures. Thus, it builds the structure from the original serial program in memory before spawning consumer loop iterations that read elements from memory. As a result, execution is serialized by operations such as schedule structure insertions. S5 enables programmers to choose among the three other flows, all of which exploit Swarm hardware’s timestamp-ordered execution model to reduce or eliminate serialization.

The parallel in-memory flow in Figure 4-2(b) optimizes a schedule structure to allow efficient parallel operations. For example, Figure 4-3 shows how a programmer may use S5 to optimize the schedule structure `valuable_widgets` by changing its type from `std::vector` to `s5::fifo`. The `s5::fifo` ADT tells the compiler and hardware that operations on `valuable_widgets` can be parallelized as long as we preserve the appearance that elements are consumed in insertion order. Figure 4-3(b) shows how the resulting parallelized execution may store each `s5::fifo` element with the timestamp of the task that inserted that element. T4+ ensures the consumer loop’s traversal of the `s5::fifo`

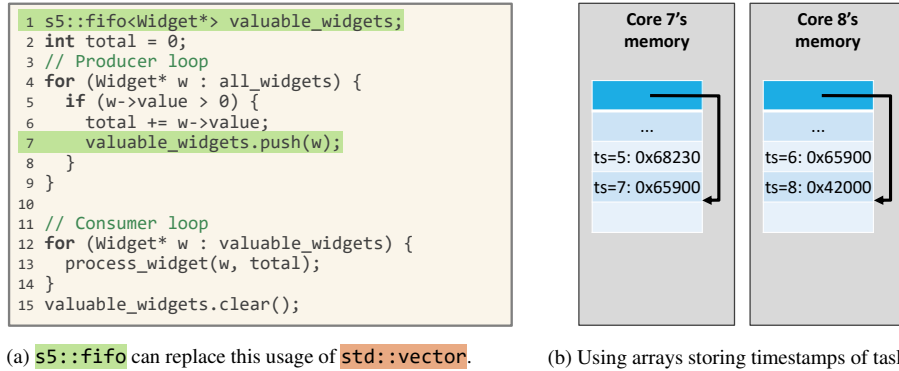


Figure 4-3: Changing two lines from Figure 4-1 is enough to use S5's parallel in-memory flow.

enforces insertion order by using the stored timestamps when spawning consumer tasks. With insertion order thus preserved, it is safe for elements to be distributed across a set of per-core shards to avoid serialization and improve locality. To further reduce data movement costs, `s5::fifo` directly passes elements to consumer tasks through registers when spawning consumer tasks.

The hardware-managed task flow shown in Figure 4-2(c) transforms schedule structures to avoid touching memory and directly insert elements into Swarm's hardware task queues (TQs) as they are produced. Specifically, we transform schedule structure insertions into task spawns. This is often simple to do because Swarm distributed TQs behave as a strict priority queue, and a strict priority order is powerful enough to implement many specific forms of scheduling. For example, Figure 4-4 illustrates how Swarm task spawns can replace `valuable_widgets` entirely. As we will see, S5 has ADTs that leverage T4+ to make creating hardware-managed tasks as easy as writing sequential code. Despite their generality, Swarm's TQs are faster than specialized software because the TQs are implemented using dedicated hardware placed near cores.

While TQs make hardware-managed tasks fast for small collections of tasks, the key limitation of hardware TQs is their finite capacity. While T4 assigns timestamps to spawners that avoid flooding TQs with tasks, we cannot stop applications from collecting large datasets. For example, if `valuable_widgets` has a billion elements, then it cannot fit on chip in hardware TQs. Swarm spills excess tasks from TQs to untracked memory using a fixed mechanism that must be sufficiently

```

1 void producer(...) {
2     int* total = new int{0};
3     // Producer loop to be automatically parallelized by T4
4     for (Widget* w : all_widgets) {
5         if (w->value > 0) {
6             *total += w->value;
7             swarm::spawn(consumer, ..., w, total);
8         }
9     }
10    swarm::spawn(deallocate, ..., total);
11 }
12
13 void consumer(swarm::Timestamp, Widget* w, int* total) {
14     process_widget(w, *total);
15 }
16
17 void deallocate(swarm::Timestamp, int* total) {
18     delete total;
19 }

```

Figure 4-4: Spawning hardware-managed tasks eliminates the need for application code to explicitly manage a dynamic data structure. S5 makes it easy to match the speed of this approach without requiring users to refactor code to use such explicit tasks spawns.

generic to handle any Swarm task. Specifically, Swarm spills tasks to memory as descriptors that match how tasks are represented in hardware queues, and stores these descriptors in software priority queues. These descriptor priority queues are both larger and far more expensive to maintain than simple arrays. As a result, if `valuable_widgets` grows huge, a programmer may need to revert back to the parallel in-memory flow using `s5::fifo` as in Figure 4-3 to avoid expensive TQ spills.

The memory-backed task flow shown in Figure 4-2(d) combines the benefits of specialized in-memory structures with the speed of hardware-managed tasks, always achieving the best performance so users need not tune the source code depending on dataset size. To do this, S5 extends hardware in a simple way to enable software to provide an application-specific backing store for spilled tasks, while hardware TQs act as a fast cache that manages recently created and soon-to-execute tasks. Tasks that fit in TQs automatically avoid invoking software handling or accessing the memory subsystem at all. Just as a hardware cache benefits general computation by combining the performance of a small dedicated memory with the capacity of a large memory address space, S5’s novel architecture benefits general scheduling by combining the performance of dedicated hardware queues with flexibility for programmers to create software libraries with efficiently tailored schedule structures.

All three S5 flows build on Swarm’s hardware mechanisms for strict

priority scheduling to simultaneously serve diverse application scheduling needs as well as to preserve easy-to-use sequential semantics. This makes S5’s ADTs easy to use as drop-in replacements for typical schedule structures in general sequential programs. S5’s guaranteed sequential semantics ensures users never face deadlock nor the need to manually decide how to explicitly parallelize or synchronize surrounding user code. S5 ADTs are safe to use in general code, without restrictions on the operations that may be performed in parallel tasks or exclusions on other sources of parallelism in a program. This is because S5 leverages and extends T4’s techniques to implicitly parallelize an *entire* program in a composable way, implicitly exploiting nested fine-grained parallelism.

In summary, S5 makes several novel contributions:

- We are the first to describe systematic optimizations to transform in-memory schedule structures for efficient parallelization that retains sequential semantics (Section 4.3.1). This makes S5 easy to use within complex software while achieving good parallelism, space usage, work efficiency, and locality.
- We enable sequential programs to schedule tasks directly using Swarm’s high-performance hardware task management (Section 4.3.2), improving the scalability of sequential programs to obtain performance competitive with manually parallelized programs.
- We define a new hardware architecture extending Swarm’s task management with application-specific semantics with almost no new hardware costs (Section 4.3.3), reusing existing hardware priority queues as a general primitive that can benefit diverse and complex programs. This allows software to specify not only application-specific task encodings but also semantic changes such as partial ordering or deduplication.
- We modify the GraphIt compiler to automatically generate code using S5 (Section 4.4). This enables translating programs from the GraphIt domain-specific language to leverage S5 without any source code changes.

We integrate S5 into complex existing software to demonstrate that S5 enables difficult-to-parallelize applications to scale linearly to 256 cores while achieving similar or better work efficiency and space efficiency than serial code. As a result, S5 achieves multiple orders of magnitude in parallel speedups and matches or vastly outperforms prior work in both automatic and manual parallelization (Section 4.5). S5 achieves all this while retaining the ease of sequential programming.

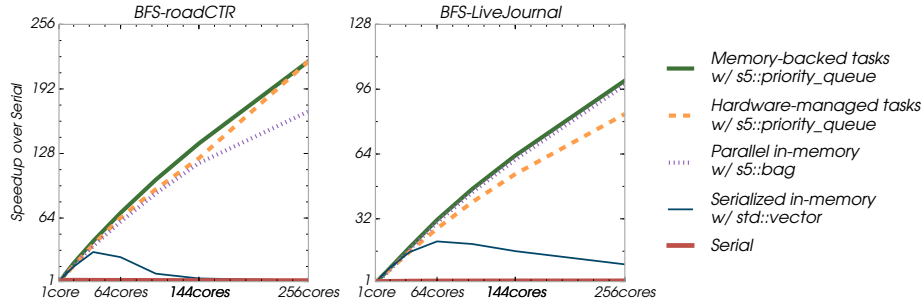


Figure 4-5: T4+’s parallel speedup for the four flows in Figure 4-2, normalized to the fastest available serial code for BFS, which stores frontiers in two raw arrays.

4.2 S5 Makes Implicit Parallelization Effective

We now show how S5 can transform schedule structures to use all three new task flows and achieve efficient scalability on hundreds of cores. Specifically, we will use the running example of breadth-first search (BFS) on a graph. BFS is representative of applications where Chapter 3 found limited parallelism due to frequent data dependences that occur when populating short-lived schedule structures.

We will examine four implementations of BFS using each of the four flows in Figure 4-2. Figure 4-5 shows the performance of these four parallelized BFS versions on two graphs with very different properties: roadCTR is a road network with few edges per vertex and high diameter, while LiveJournal is a social-network graph with low diameter, where some vertices have hundreds or thousands of incident edges. All three S5 flows achieve much better scalability than the serialized in-memory flow. Between the parallel in-memory flow and the hardware-managed task flow, neither consistently outperforms the other. But the memory-backed task flow enabled by S5’s new architecture always achieves the best performance. We now explain how S5 enables each of these flows.

4.2.1 Optimized in-memory flow for intra-level parallelism

A breadth-first graph traversal must schedule its visits to nodes in increasing distance order from the starting node. To do this, the code shown in Figure 4-6 uses in-memory schedule structures that track the membership of vertices in *frontiers*. All nodes in one frontier are visited before the nodes in the next frontier. Figure 4-6(b) shows how easy it is to replace `std::vector` with `s5::bag`, an ADT that conveys to

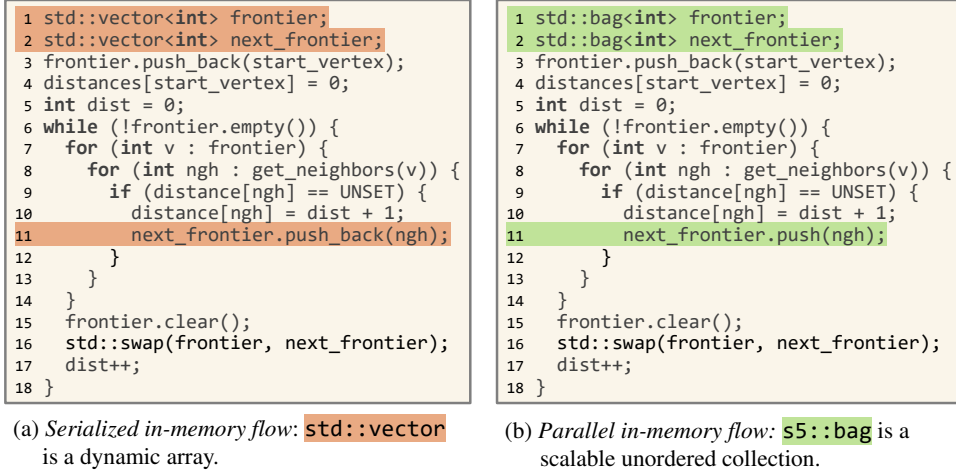


Figure 4-6: S5’s ADTs can be used as drop-in replacements for a serial scheduling structure in breadth-first earch (BFS).

the system that nodes within one frontier may be visited in any order. Figure 4-5 shows that the optimized `s5::bag` schedule structure can scale up to 256 cores, while `std::vector` only scales to tens of cores at best, chiefly due to insertion serialization.

Unfortunately, a BFS implementation that maintains two frontiers will still impose extraneous serialization regardless of the choice of data type for each frontier, which will limit performance on high-diameter graphs. This is because the frontiers are recycled as the algorithm moves from one distance level to the next. This serialization manifests as data dependences involving the swapping of the two frontiers on line 16 of both code versions in Figure 4-6. The swap is a cheap operation that just moves some pointers, but all insertions and traversals use these same pointers, so data dependences prevent tasks ordered after the swap from speculatively executing in parallel with tasks ordered before the swap. Thus, serialization occurs between iterations of the outer **while** loop regardless of whether frontiers have type `std::vector` or `s5::bag`. This serialization also occurs in all prior work that parallelizes BFS without speculation, which is done using bulk-synchronous level-by-level parallelism [19, 135, 268]. This approach works well for the low-diameter LiveJournal graph. By contrast, roadCTR has many distance levels, so the serialization that occurs between levels limits scalability.

```

1 s5::priority_queue<int> frontiers;
2 frontiers.push_init(/*key=*/0, start_node);
3 distances[start_vertex] = 0;
4 frontiers.for_each([](uint32_t key, int v) {
5     for (int ngh : get_neighbors(v)) {
6         if (distance[ngh] == UNSET) {
7             distance[ngh] = key + 1;
8             frontiers.push(key + 1, ngh);
9         }
10    }
11 });

```

Figure 4-7: *Hardware-managed or memory-backed task flow:* A single priority queue subsumes all frontiers. The call of `for_each()` on line 4 uses a lambda to express the task to consume one element.

4.2.2 Hardware is fast and unlocks inter-level parallelism

We can improve performance by replacing in-memory schedule structures with dedicated hardware that avoids unneeded dependences and quickly delivers elements waiting to be consumed. We can do this using the hardware-managed task flow in Figure 4-2(c) if we can translate the application to fit Swarm hardware’s semantics. This turns out to be easy because Swarm provides the semantics of a strict min-priority queue, guaranteeing low-timestamp tasks appear to execute before higher-timestamp tasks. Under the hood, Swarm executes tasks in parallel so task order does not restrict parallelism. To make this capability easily accessible to sequential programs, S5 provides the `s5::priority_queue` ADT, which is implemented as a thin wrapper around directly spawning tasks to hardware, using each integer priority key specified by the application as a timestamp. The `s5::priority_queue` ADT can be compiled to run on Swarm or to S5’s new architecture.

When `s5::priority_queue` is compiled to run on Swarm, it uses the hardware-managed task flow. We can easily rewrite many schedule structures to use a strict priority queue, making a priority queue a very general primitive. For example, Figure 4-7 shows how BFS can use a priority queue with distance levels as priorities, ordering lower distances before higher distances. The call to `push()` on line 8 compiles into a single `spawn_task` instruction that spawns a Swarm task scheduled to run in the next distance level.

If tasks fit in hardware TQs, `s5::priority_queue` will be efficient. Enqueues and dequeues each require only a single instruction to access TQs placed near cores, each of which implemented with a pipelined heap

that can sustain one insertion and one dequeue every cycle [27, 101]. `s5::priority_queue` also eliminates serialization between priority levels, allowing task with different priorities to run speculatively in parallel. As a result, Figure 4-5 shows the hardware-managed task flow achieves good performance on the road graph, where the number of nodes per level is small enough to fit in TQs in a 256-core Swarm system.

4.2.3 The memory-backed task flow combines the benefits of prior flows

However, LiveJournal has hundreds of thousands of elements queued in a single distance level, exceeding the fixed capacity of hardware TQs. Full TQs spill task descriptors to memory, incurring the costs of a generic software priority queue. These generic priority queues are unnecessarily costly and a poor fit for BFS. They use a descriptor format that uses dozens of bytes per task to store a function pointer, timestamp, five more 64-bit register values, and additional flags. Additionally, a priority queue of size N will have enqueue or dequeue operations that require $\Theta(\log N)$ dependent comparisons.

Therefore, while hardware support can make a generic priority queue very efficient for collections small enough to fit, large collections benefit greatly from more specialized schedule structures. This motivates S5's novel architecture, which will enable the memory-backed task flow in Figure 4-2(d) where software handles TQ spills with application-specific optimizations. The BFS using `s5::priority_queue` will automatically use memory-backed tasks when compiled for S5's new architecture. For BFS, memory-backed tasks can exploit the fact that insertions are not arbitrarily ordered: each insertion always goes only to the next priority level, enabling S5 to store spilled tasks just as they would be stored by a pair of `s5::bags`.

Figure 4-5 shows that memory-backed tasks always give the best performance for BFS. They provide the speed and parallelism of hardware-managed tasks for small task collections in RoadCTR. They also provide the efficiency of a simple in-memory schedule structure customized to BFS's needs for large collections in LiveJournal.

When we explore using S5 in other applications, we will see the ability for software to customize queue semantics will enable further performance optimizations that exploit the specific needs of other applications as well, such as needs for particular forms of ordering or

Method	Description
void push(ElemTy x)	Insert x into the collection. May not be called during execution of <code>for_each()</code> .
void for_each(UnaryFn f)	Apply function object f to each element.
size_t materialize(ElemTy* dst)	Write all elements to an array in memory starting at dst, then return the number of elements.
void clear()	Discard all elements, emptying the collection.
bool empty()	Check whether the collection is empty.
void reserve(size_t capacity)	Pre-allocate memory to make push() faster.

Table 4-1: API of `s5::fifo`, `s5::lifo`, and `s5::bag`, all of which store a collection of elements in memory.

deduplication of redundant tasks.

4.3 S5 Implementation and Optimizations

We now detail how the implementation of S5 optimizes in-memory schedule structures for parallelism and locality, safely enables software to skip using memory with hardware-managed tasks, and allows software to specialize the semantics of memory-backed tasks.

4.3.1 Implicitly parallelizable in-memory structures

The simplest S5's ADTs are those that use the parallel in-memory flow. These ADTs are the `s5::fifo`, `s5::bag`, and `s5::lifo`, which all exploit hardware support for timestamp-ordered tasks with a general strategy that divides the structure into *shards*. Each shard is owned by one core that accesses that shard to exploit locality, and the collection of shards enables parallel operations.

Table 4-1 shows these types support inserting elements, traversing the collection, and other operations. These S5 ADTs do *not* provide fast random access to their elements: it is only possible to access elements by traversing the entire collection in an order determined by the particular ADT. `s5::bag` is an unordered collection, `s5::fifo` orders elements in insertion order, and `s5::lifo` orders elements in reversed insertion order. Each S5 ADT has a method named `for_each()` to perform a traversal. `for_each()` takes one argument: a callback (usually expressed as a C++ lambda) specifying the computational task to be performed for each element. T4+ also provides syntactic sugar so traversals may be expressed as a range-based **for** loop:

for ([declaration] : [collection]) { [body] }

When such a loop acts on a collection that has a S5 ADT for its type, the loop syntax is automatically translated¹ into a call to `for_each()`:

[collection].`for_each`([=]([declaration]) { [body] });

Figures 4-3 and 4-6(b) demonstrate the use of the loop syntax.

4.3.1.1 Compiler and hardware optimizations for schedule structure traversals:

As some applications spend a significant amount of time performing operations on schedule structures, it is important to tune frequent reads and writes to the shards to be very efficient. To better exploit locality when collections are small, we extend the Swarm ISA so the `spawn_task` instruction has a flag indicating whether a hint is an ordinary spatial hint that localizes the task to a tile, or a *per-core spatial hint* that specifies a specific core the task must run on to maximize L1 cache hits. Each ADT's implementation can then use per-core hints to localize both reads and writes to shards. To avoid starving tasks with per-core hints, S5 must also adapt Swarm hardware to abort higher-timestamp tasks that might be stalled while occupying a core when a lower-timestamp task with a per-core hint reaches the head of the TQ.

The new T4+ compiler offers capabilities to allow automatically parallelized code to safely call code that was manually parallelized directly using Swarm's low-level API (Section 2.3.2). This allows low-level programmers to write manually parallelized subroutines that spawn their own tasks at multiple timestamps or even create Fractal domains. To do this, low-level programmers must tag the declarations of functions that are manually parallelized. Then, T4+ ensures each call from an automatically parallelized function to a manually parallelized function will be spawned as its own task. This makes it safe for the manually parallelized function to create its own Fractal domain and specify any timestamps it needs for tasks in its new domain.

This capability is used to implement carefully optimized parallel traversal algorithms in `for_each()`, which are specialized to the strategies that each ADT uses to handle their ordered or unordered elements.

¹ Our current implementation forbids loop bodies containing **break** statements. Future work could handle this by automatically transforming the control dependence between loop iterations into a data dependence using predication [8] on a new termination flag variable, just as was done in the case of progressive loop expansion (Section 3.6.1).

One feature all implementations of `for_each()` have in common is an optimization that reduces data movement costs. Rather than having worker tasks read the elements they should consume from memory, we have their parent task, a spawner, read several values at once to exploit spatial locality and then pass the values to consumer tasks through registers. This allows the spawner to be given an appropriate spatial hint to localize shard accesses while the consumer task may have a separate automatically generated hint (Section 3.7.2) to improve locality.

Each consumer task, which runs the application-provided callback passed to `for_each()`, is still automatically parallelized. S5 uses Fractal domains to safely compose fine-grained nested parallelism within and across invocations of this parallelized callback. We ensure S5 spawns a separate task to consume each element of the collection to make it safe for the automatically parallelized callback to create its own domain where it may use T4's automatically generated timestamps to preserve sequential semantics, so the consumer task appears to execute as one atomic unit.

4.3.1.2 In-memory implementation of individual ADTs

s5::bag: Each *shard* of an `s5::bag` uses a dynamic array to hold unordered elements. We ensure each core only inserts elements into its own private shard, avoiding cache ping-ponging and serialization of tasks on different cores. During a traversal, we exploit the flexibility to visit elements any order by choosing to interleave the timestamps for tasks reading from different shards to traverse all shards in parallel at a balanced rate.

s5::fifo: Unlike `s5::bag`, `s5::fifo` needs to record the order of insertions. `for_each()` uses stored order information to assign timestamps to consumer tasks. To do this, T4+ performs static analyses to select between two strategies for each `s5::fifo` instance. The first strategy is to explicitly store an integer timestamp when each element is inserted. The second strategy is to use per-core hints to specify which shard insertions should go to so that the position of an element in a shard implicitly encodes its order relative to other elements.

For the first strategy, which stores timestamps explicitly in memory, `s5::fifo` can obtain timestamps from the tasks that perform insertions by invoking a T4+ compiler intrinsic. This intrinsic returns not the

timestamp of the task that calls `push()`, but the timestamp that orders the loop iterations that enclose the insertions, which may be in a higher Fractal domain. For a T4-parallelized loop, this will equal a small integer multiple of the loop index (Section 3.5.3). This strategy can be applied in cases such as Figure 4-3.

In cases such as BFS where nested loops perform insertions into an `s5::fifo`, we use a more complex strategy. This is because it is expensive to re-encode timestamps from multiple Fractal domain levels into a wide integer timestamp that orders all insertions. In this case, `s5::fifo` creates a sequence of shards and ensures each shard in the sequence contains elements that are ordered before the elements in the next shard, using just the timestamp from the nearest outer loop that encloses all insertions to assign insertions to shards. Later, when `for_each()` performs its parallel traversal, it first computes a prefix sum over the sequence of shard sizes, in order to obtain the timestamp offset at which each shard will begin spawning the deferred application tasks with consecutive increasing timestamps so that order of the deferred tasks matches insertion order.

s5::lifo: The implementation `s5::lifo` is identical to `s5::fifo`, except that the order of timestamps is reversed. Any consumer that `s5::fifo` would spawn with timestamp `t`, `s5::lifo` spawns with timestamp `UINT64_MAX - t`.

4.3.1.3 Additional optimizations

Each ADT supports checking if the collection is empty in constant time by maintaining a single flag set to true if the collection is non-empty. The first insertion sets this flag, and further insertions read this flag and see it is already set, resulting in shared copies of this flag reaching L1 caches so most reads are cheap. T4+ performs a static analysis to eliminate this flag entirely in any instance where we can determine `empty()` is never called.

Each ADT exploits T4's support for efficient task spawns to improve load balance and reduce abort costs (Section 3.2.2) by using tiny tasks to perform insertions. Rather than having `push()` immediately insert into a shard, it spawns a task with no spatial hint to perform the insertion, so that the insertion executes on a random core, without any hint-based serialization that the parent task might be subject to. This is accomplished using T4's task annotations (Section 3.5.2) within the

Method	Description
void push_init(uint64_t key, ElemTy x)	Insert the pair (key, x) with priority key. May not be called during execution of <code>for_each()</code> .
void for_each(BinaryFn f)	For each element, apply function object f to it and then discard it.
void push(uint64_t key, ElemTy x)	Insert the pair (key, x) with priority key. May only be called during execution of <code>for_each()</code> .
void reserve(size_t capacity)	Provide a hint for custom spillers to avoid frequent dynamic memory reallocation.

Table 4-2: `s5::priority_queue` allows insertions during a parallel traversal.

S5 library implementation.

4.3.2 Directly using hardware-managed tasks

S5 provides additional ADTs to enable the hardware-managed task flow in Figure 4-2(c), which can directly insert tasks into hardware TQs rather than storing tasks in memory. `s5::priority_queue` is the simplest ADT to enable this flow. Table 4-2 shows the API of `s5::priority_queue`. `s5::priority_queue` allows consumer tasks executing within `for_each()`'s parallel traversal to call `push()` to insert more elements in the midst of a traversal. Each call to `push()` directly inserts one element into Swarm's hardware TQs in the form of consumer task that takes the element as an argument. This skips storing elements to memory and exploits the speed of dedicated hardware TQs.

As with other S5 ADTs, T4+ composes fine-grained nested parallelism across different `s5::priority_queue` consumer tasks, even those with different priorities. To do this safely, T4+ must spawn all consumer tasks in a single Fractal domain so that using the application-specified priority as a timestamp will allow hardware to correctly order consumer tasks. We still use T4's techniques to automatically parallelize the application-supplied callback that consumes a priority queue element, exploiting fine-grained parallelism among nested loop iterations or function calls. T4 creates nested subdomains to assign timestamps that order these nested tasks (Section 3.5.3), preserving sequential semantics while allowing Swarm to spawn tasks in parallel.

The challenge of implementing `s5::priority_queue::push()` is that its callsite may be arbitrarily deeply nested within a hierarchy of subdomains created by T4. So each call to `push()` must know how many

domain levels up it must traverse back up the domain hierarchy to reach the domain where the consumer tasks are ordered by application-specified priorities. T4+ therefore performs an analysis of the program’s call graph to identify all functions that could be directly or indirectly called within `for_each()` and that could directly or indirectly call `push()`.² T4+ then rewrites these functions to dynamically compute their relative domain depth at runtime. To do this, T4+ adds an integer argument to these functions indicating the domain depth at which the function is called, relative to the application-priority-ordered domain. Then, at each callsite to these functions, a static analysis determines an appropriate domain-level offset based on how deeply nested the callsite is in its enclosing function’s loop nests, and this offset is added to the function’s domain-depth argument before being passed to the callee. This is a local, selective change to the calling convention much like T4 already performs when it selectively transforms functions into continuation-passing style (Section 3.4).

This transformation ensures that each `s5::priority_queue::push()` call knows the relative domain depth at which it is called, but it must then use this information to place a consumer task in the correct domain. Since Swarm’s Fractal execution model only allows a task to spawn a child at most one level higher in the domain hierarchy, `push()` might not directly spawn a consumer, instead spawning a chain of zero or more spawner tasks that each spawn their child one domain level higher to traverse up the domain hierarchy. The final spawner in the chain spawns the consumer task to the correct domain.

Finally, we modify the compiler and hardware to prevent performance pathologies involving spills for superdomain-traversing task chains. We observe that T4 already nicely avoids unneeded spills of tasks in deeply nested subdomains because TQs prioritize tasks for dequeue by virtual time, running tasks at the deepest domain before expanding task subtrees in other domains.³ But chains of spawners traversing up the domain

² To ease this static analysis, T4+ currently presents the user with a compiler error if `s5::priority_queue::push()` is called by a function with external linkage. Future work could better support composition of separately compiled code by performing the static analysis during link-time optimization or by changing the calling convention used across all translation unit boundaries to always pass domain depth information.

³ This can be compared with a low-level optimization used in Swarm that relies on programmers manually tagging tasks that are likely to spawn multiple children with a `PRODUCER` flag. Hardware prioritizes `PRODUCER`-flagged tasks after other same-timestamp tasks to avoid creating more tasks before they are needed. T4’s use of

hierarchy are forced by their different domains to be spread apart in virtual time order, making some steps within the chain likely to linger in TQs and need to spill. Since running and committing a task that spawns no more than one child does not increase the total number of tasks in the system, we prefer to run tasks within a chain of spawners instead of spilling them, so we can spill just the single consumer task at the end of the chain if needed. We wish to avoid the redundancy of spilling a spawner that later spawns the consumer which spills again.

To accomplish this, we use two mechanisms to discourage unneeded spills of spawners within chains. First, T4+ assigns no timestamp to spawner tasks that only spawn a single task to a parent domain. When a non-timestamped task's parent has committed, hardware prioritizes the non-timestamped task for dequeue as if it had timestamp zero within its domain, in order to run and commit it quickly instead of spilling it. Second, T4+ ensures each task in a spawner chain inherits its parent's spatial hint so it stays local to its parent's tile. This reduces communication costs and also makes it likely that the TQ slot a spawner vacates when it runs is available to hold its single child. T4 may still automatically assign an independent spatial hint to the consumer task spawned at the end of the chain.

4.3.3 Spilling tasks to application-specific backing stores

`s5::priority_queue` also enables the memory-backed task flow in Figure 4-2(d). Swarm's hardware TQs are very efficient for a collection that fits within their finite capacity, but they spill large collections to an application-agnostic in-memory structure that holds generic task descriptors. We can enhance `s5::priority_queue`'s performance on large collections by replacing Swarm's generic spill mechanism with one that allows software to manage the backing store for spilled tasks, giving us the ability to encode large datasets compactly and with behaviors customized for an application's specific needs. This requires changes to both hardware and software. We now first explain S5's new hardware-software interface, then show how `s5::priority_queue` and T4+ use this new interface, before finally explaining S5's hardware

Fractal has similar benefits, because T4 ensures any loop that may create many tasks will always create a subdomain in which the tasks are placed, and Fractal ordering favors running tasks within a subdomain before parent-domain tasks that may create more domains.

implementation.

4.3.3.1 The S5 ISA extension

We change hardware to store a software-controlled mapping from task functions (code pointers) to matching software routines called *custom spillers*. S5 extends the Swarm ISA with a new `register_spiller` instruction that adds or updates a mapping entry for one task function to map to a matching custom spiller. This new instruction may only run non-speculatively. Any speculative task that runs it incurs an exception, which forces the task to wait until it can run non-speculatively (Section 2.3.4). When hardware needs to spill tasks that will execute a particular task function, hardware may run a matching custom spiller in place of the generic spiller that stores tasks in the form of descriptors. All custom and generic spillers execute non-speculatively, can access a dedicated region of untracked memory without any conflict checking or aborts, and can only spill tasks whose parents have committed. But when a custom spiller executes the existing `spill_task` instruction, hardware only returns tasks with task functions that match the executing spiller.

When compiling for S5's new architecture, T4+ enables support for `s5::priority_queue` to register a custom spiller during each call to `for_each()`. Specifically, `s5::priority_queue::for_each()` starts by spawning a dedicated task to register a custom spiller, using compiler support to obtain the pointer to the optimized wrapper task function T4+ generates that calls the application callback. Before it returns, `s5::priority_queue::for_each()` spawns another task to deregister the custom spiller (by running `register_spiller` again with a null spiller pointer).

4.3.3.2 Software-controlled backing stores

When it runs, a custom spiller must dynamically build scheduling structures to store tasks in untracked memory. These in-memory structures act as a backing store, while hardware TQs function as a fast cache. Custom spillers receive descriptors consisting of timestamps, arguments, and other metadata that `spill_task` makes available in registers, so software can decide how or whether to store these tasks in a format specialized to the behavior of the application. Custom spillers may also spawn new tasks to *requeue* tasks from memory to the local hardware TQ. Requeuing a task means removing it from untracked memory

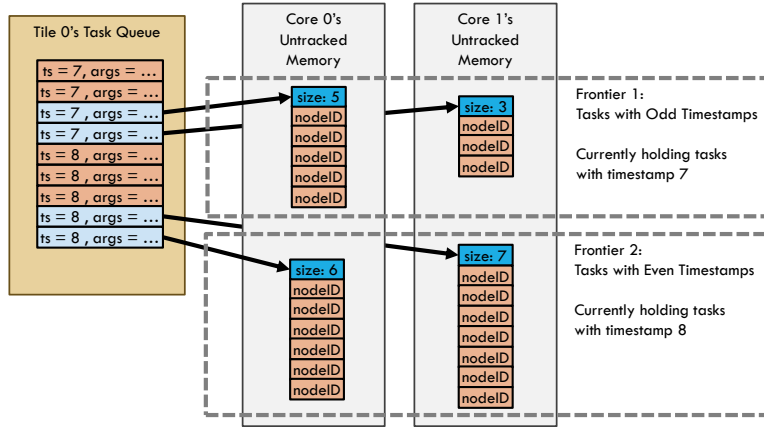


Figure 4-8: A snapshot of memory-backed tasks waiting to execute for the BFS code in Figure 4-7. **Consumer** tasks are either resident in hardware TQs or spilled to two sets of per-core arrays holding just node IDs. A 4-byte ID uses $16\times$ less memory than a 64-byte task descriptor.

and spawning it back to the local hardware TQ using the ordinary `spawn_task` instruction.

Custom spillers and requeuers may be written by the application programmer. S5's runtime library also provides various built-in custom spillers and requeuers that will be used by default if the application code does not provide its own custom spiller. T4+ uses type information and hints from the programmer to select between these default built-in custom spillers.

We now present two ways S5's built-in custom spillers encode tasks compactly in memory, and then we show how support for custom spills is extremely simple and cheap to implement in hardware. When we evaluate S5 on more applications in Section 4.5, we will show how application programmers can provide custom spillers with additional specialized behaviors, such as task deduplication, which provides enormous benefits for particular applications.

First, for the BFS implementation in Figure 4-7, we exploit the fact that newly spawned consumer tasks always have timestamp exactly 1 greater than the most recently dequeued consumer task. We can therefore group spilled tasks by timestamp, forming two frontiers as shown in Figure 4-8. This does not constrain parallelism, as TQs may hold speculative tasks with higher timestamps. Within one group or frontier, spilled tasks share a single timestamp value and may run in

any order. Therefore, the elements are stored exactly like they are in an `s5::bag`, distributed into per-core shards to enable efficient parallel operations. Each task is represented by storing only the element the task will consume. So for BFS we store only 32-bit vertex IDs. But unlike when an application calls `s5::bag::push()`, a custom spiller does not distribute tasks to other tiles for load balancing, instead performing insertions directly into the local shard owned by the core on which the spiller runs. This ensures spilled tasks respect their original spatial hints and do not move between tiles.

When inserting the first element into an empty shard, a custom spiller spawns a custom requeuer task with a pointer to the shard. This custom task requeues several (e.g., up to 7) tasks from the in-memory shard to the local hardware TQ. The requeuer task finishes by spawning another requeuer task for the same shard if the shard is still non-empty. In this scheme, each shard will have at most one requeuer in its local TQ at a time. These requeuer tasks are given per-core spatial hints that force the requeuer to run on the same core the shard belongs to. This ensures there are never concurrent spillers and requeuers accessing the same shard, avoiding cache ping-ponging and any need for explicit synchronization.

When applications need to spill tasks with a wide range of timestamps (unlike BFS, where two timestamps suffice), `s5::priority_queue` defaults to a custom spiller that stores (priority, value) pairs in memory, implementing a memory-backed priority queue. When using this class of custom spillers, T4+ analyzes type information in the application program to select a custom spiller to store elements compactly. For example if the types used in an application callback (like that in Figure 4-7) indicate priority keys always fit in 32 bits, there is no need to store larger 64-bit timestamps. Each execution of the custom spiller will write several (priority, value) pairs to a chunk of untracked heap memory and then spawns a custom requeuer task that is responsible for reading and deallocating the memory chunk as well as requeuing the enclosed tasks to the TQ.

Whenever we register this sort of custom spiller that could create unboundedly many requeuers by running repeatedly, we also register an associated *recursive* custom spiller to spill these particular requeuer tasks. This recursive spiller represents each spilled requeuer by storing only its single pointer to the chunk it will requeue from. When the recursive spiller spawns a requeuer, it uses the same requeuer task

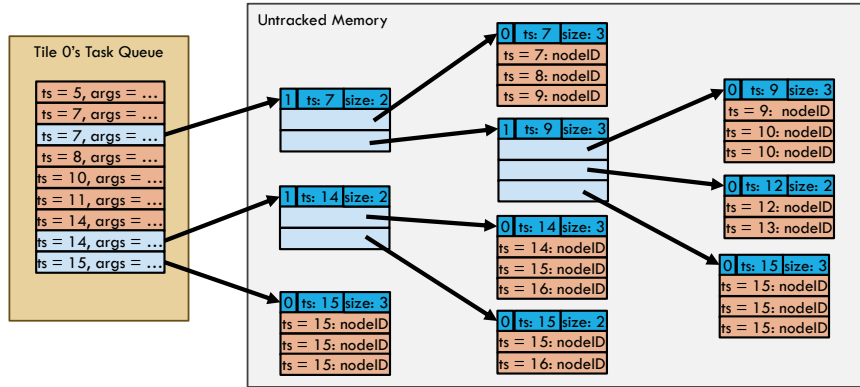


Figure 4-9: A snapshot of memory-backed tasks that consume nodes of a graph from a general priority queue. Spilled **consumer** tasks are stored in memory as (priority, node ID) pairs. With 4-byte priorities and node IDs, each pair is only 8 bytes, 8× smaller than a 64-byte task descriptor. Each memory chunk is owned by a **requeuer** task, which may reside in the TQ or be encoded in memory as a pointer to the chunk it owns.

function as the non-recursive spiller of consumer tasks. This enables the recursive spiller to be invoked again to spill the very same requeuers it generates.

Figure 4-9 shows each memory chunk holding spilled tasks also has a header containing the timestamp of the requeuer pointing to the chunk, the number of spilled tasks in the chunk, and a flag indicating if the spilled tasks are requeuers or consumers. This in-memory structure is similar to that created by Swarm’s generic spill mechanism, but it is more compact because it is aware of the types of the data elements among each task’s arguments that need to be stored, and it avoid redundantly storing other metadata such as function pointers and flags. It also segregates spilled consumers and requeuers to allow the spillers and requeuers that access these memory chunks to use separate highly tuned tight loops to handle the different spilled task types.

We ensure each requeuer runs and requeues tasks only when the TQ needs them. To do this, we first ensure tasks within a single memory chunk are sorted by timestamps, so a requeuer requeues lower-timestamp tasks before higher-timestamp tasks. Each requeuer is then given timestamp equal to that of the first task it will requeue. The TQ uses this timestamp when prioritizing tasks for dispatch or spills. This ensures the requeuer executes only when it will requeue a spilled task

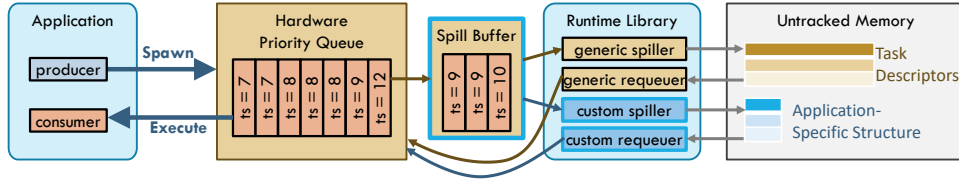


Figure 4-10: Hardware launches custom spills using logic local to each hardware tile’s small spill buffer. No change is needed either in Swarm’s large hardware priority queues, nor in the protocols used to spawn tasks and manage speculation.

that needs to run soon.

4.3.3.3 S5 hardware implementation

Figure 4-10 shows the key hardware structures and software tasks types which facilitate spills. S5 makes very little change to the hardware that was already needed to implement Swarm’s generic spills. We now briefly summarize how the microarchitecture of a Swarm TQ handles generic spills before describing how we extend Swarm hardware to support custom spills. More details may be found by examining the open-source Chronos FPGA prototype [1].

Swarm manages each TQ’s idle tasks as a double-ended priority queue ordered by timestamps, in order to support dispatching low-timestamp tasks for execution and removing high-timestamp tasks for spills. (The double-ended priority queue may implemented in hardware with a variation on a binary heap called a min-max heap [16], which can be pipelined just like an ordinary single-ended heap [27, 101].) This prioritization results in intelligent eviction decisions: TQs only spill tasks spawned far enough in advance of when they will execute to be crowded out by later-spawned tasks that need to stay in the hardware TQ to execute sooner. When TQ occupancy passes a threshold, the TQ puts several (e.g., 15) tasks into a hardware *spill buffer* and then executes a generic spiller that takes all tasks from the spill buffer and stores their descriptors to untracked memory. The spill buffer allows spillers to exploit how tasks taken from the max-timestamp end of the priority queue are guaranteed to be sorted in decreasing timestamp order. This makes it easy for the spiller to write the descriptors within a spilled chunk of 15 tasks in a sorted order to make the lowest-timestamp

task quick to find.

The only hardware structure that needs to change to implement custom spills is the small spill buffer. `register_spiller` is implemented by broadcasting the new registration to all tiles, where spill buffers store just the few (e.g., four) most recently registered (task function, custom spiller) pairs. When a full TQ decides to move tasks to its spill buffer, the spill buffer checks each task it receives against the registered custom spillers to build a linked list of matching tasks for each custom spiller. Storing these linked lists requires adding only four bits to each spill buffer entry to store a next-node index. If no list contains at least two matching tasks among the 15 spill buffer entries, we run the generic spiller which retains its behavior from Swarm, removing all 15 tasks from the spill buffer. Otherwise, we execute the custom spiller with the longest list, and each execution of `spill_task` removes one task from that list. When the custom spiller finishes, remaining tasks in the spill buffer are returned to the priority queue. The hardware changes to implement all this are extremely cheap. To support the worst case of spillers running concurrently on every core, we need only one spill buffer per core, which holds only 15 tasks.

4.4 Fully Automatic Parallelization of Graph Algorithms

So far, we have described how S5’s ADTs enable a user to help the T4+ compiler effectively parallelize sequential programs. But a compiler with sufficient knowledge of application semantics can also exploit S5 fully automatically, without any manual effort. We demonstrate this by automatically parallelizing algorithms written in GraphIt. GraphIt [270, 271] is a domain-specific language designed to make it easy to generate state-of-the-art parallel implementations of a wide variety of graph algorithms. Graph processing is a critical computational domain to study as it is at the heart of many applications [38, 60, 172, 209, 258] but is notoriously difficult to optimize for parallel execution [144, 271] due to highly irregular memory access patterns that often saturate memory bandwidth or cause poor utilization of parallel hardware.

We collaborated with the developers of GraphIt to combine the capabilities of GraphIt, T4, and S5 in a single framework called UGC [34] that makes it easy to support today’s rapidly expanding variety of

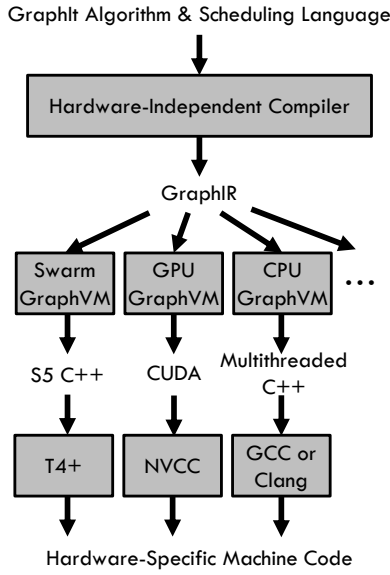


Figure 4-11: The Unified GraphIt Compiler framework (UGC) decouples hardware-independent analyses and transformations that capture high-level knowledge of application semantics in GraphIR from hardware-dependent GraphVMs and backend compilers.

hardware architectures. Focusing on one domain, graph analytics, makes it feasible for UGC to leverage understanding of high-level programmer intent to safely perform hardware-independent and hardware-specific optimizations that exploit knowledge of the problem domain. UGC not only understands the format of input graph datasets, it knows which dynamic data structures in a program represent frontiers (subsets of vertices) in the graph. UGC leverages the restricted operations programmers are allowed to use to access data structures in the GraphIt high-level language to identify which program statements populate or consume frontiers and determine when data accesses need ordering or atomicity.

Figure 4-11 shows how UGC compiles and optimizes GraphIt programs in three phases. First, hardware-independent static analyses and transformations are preformed to generate a new domain-specific intermediate representation called GraphIR that captures high-level program structure and serves as an interface to different hardware backends. Then, GraphIR is lowered into code for a particular architecture using a GraphVM, which performs a set of architecture-specific compiler passes and makes decisions about how to store vertex subsets, for example by

using optimized in-memory software-managed work queues. Finally, a backend compiler such as T4+ generates the machine code for parallel execution.

We have developed a Swarm GraphVM within UGC that can emit highly optimized sequential C++ code with automatically inserted usage of S5 ADTs as well as T4 task-splitting annotations (Section 3.5.2). T4+ can then compile this annotated C++ code into highly parallel fine-grained Swarm tasks. An early version of S5 that supported the parallel in-memory flow and hardware-managed task flow, but not the memory-backed task flow, was previously demonstrated to allow UGC matches state-of-the-art performance optimizations for a wide variety of graph algorithms [34].

UGC can emit optimized code for highly different architectures without requiring programmers to rewrite any portion of existing source code written in the high-level GraphIt algorithm specification language. This source language uses program variables that hold at most a single frontier at a time, and includes operations that produce or consume elements of these frontiers, much like the serial C++ code in Figures 4-1 and 4-6. This source code is not written specifically for any target architecture, and separate GraphVMs enable the same algorithm source code to achieve state-of-the-art performance across Swarm as well as conventional CPUs and GPUs [34]. Each GraphVM provides a *scheduling language* that expresses a large space of possible combinations of program transformations and optimizations which do not to change the semantics of the program. This allows users choose either to manually specify the precise optimizations they wish to enable, or to have an autotuners to search the space to automatically find a good combination of optimization options [10, 35, 270, 271].

The Swarm GraphVM can automatically generate optimized sequential C++ that uses S5 ADTs. For example, a user can use the scheduling language to choose whether to use an ADT such as `s5::bag` to represent each frontier, or whether to automatically rewrite iterative algorithms acting on separate frontier variables into hardware-managed tasks using `s5::priority_queue`. The Swarm GraphVM automatically infers which program statements need to be grouped into callbacks passed to `for_each()`, expresses ordering information through priority values, and automatically chooses the type of custom spiller to enable efficient memory-backed tasks.

In fact, this Swarm GraphVM can automatically generate the three

versions of BFS that used S5 to achieve linear scalability on hundreds of cores in Section 4.2. These results came from compiling existing BFS code taken from evaluations of previous GraphIt compilers for conventional CPUs and GPUs [35, 271], and we make no source code changes to enable UGC’s novel optimizations.

A crucial optimization the Swarm GraphVM performs to make `s5::priority_queue`’s hardware-managed task flow particularly effective is carefully encoding the value of scalar variables such as outer loop indices or induction variables either as part of the priority key or embedded within the actual element value, which may be a struct instead of a scalar. This allows tasks to pass these values to each other directly through registers as part of their timestamp or arguments, eliminating what would otherwise be a shared global variable whose updates would be a barrier preventing speculative parallelization across outer loop iterations. For example, in Figure 4-7, the values stored to the `distance` array are rewritten to be computed from the local priority `key` rather than using a shared variable that is incremented on each outer loop iteration as in Figure 4-6.

4.4.1 `s5::levels_queue`

To ease the use of `s5::priority_queue` for particular application patterns, we provide adaptor classes that wraps `s5::priority_queue` with a more convenient interface. One such adaptor used extensively by the Swarm GraphVM is the `s5::levels_queue` ADT. When optimizing an algorithm to use hardware-managed tasks the Swarm GraphVM may generate C++ code that uses `s5::levels_queue` instead of directly using `s5::priority_queue` if an algorithm alternates between operations on subsets of vertices and operations on the entire graph. Unlike `s5::priority_queue`, `s5::levels_queue`’s `for_each()` method takes two callbacks instead of one to define what work needs to be done when traversing and consuming a collection. The first is the usual callback that defines the computation to be done per element (e.g., when consuming a vertex subset), while the second callback is guaranteed to be called exactly once immediately after each non-empty priority level. So for graph algorithms, the first callback may express actions applied to a subset of vertices, while the second callback can perform some action that processes the entire graph [34]. As is the case with callbacks for all other Swarm ADTs, both callbacks passed to

`s5::levels_queue::for_each()` may be automatically parallelized by T4+ to extract nested fine-grained parallelism.

`s5::levels_queue` is implemented as a simple wrapper around `s5::priority_queue` where every application-specified priority is multiplied by two, so that element-consuming tasks have even priority values, leaving odd priority values available to execute the between-levels callback. To ensure this callback executes after any level in which there were any data-processing tasks, every call to `push()` that spawns an element-consuming task implicitly spawns a between-level task as well. At each priority level, only the first between-level task to run actually performs the between-level callback, while the other between-level tasks do nothing. `s5::levels_queue` uses a scalable auxiliary data structure to track what priorities have already had a between-levels callback execution. Specifically, a small array in memory (e.g., 256 elements in our implementation) holds integers that record the most recent priority for which between-level tasks have run. Each between-level task indexes into this array with its priority modulo the array size and, only if the priority does not match, it increases the priority to match and executes the application's between-level callback. As we will see, `s5::levels_queue` is not only useful for compiling GraphIt programs, it also eases manual refactoring of parallelizable sequential programs.

4.5 Evaluation

We evaluate S5 on diverse applications to demonstrate how S5 makes difficult-to-parallelize applications scale to hundreds of cores, all while remaining easy to use.

4.5.1 Hardware simulation methodology

We modify the same Swarm simulator used for T4's evaluation to model S5's new architectural extensions for custom spillers and per-core spatial hints. As before, we report performance relative to serial code compiled with `clang++ -O3` on Swarm and S5 processors of varying sizes, varying aggregate queue and cache capacities in proportion to the number of cores. We simulate systems of up to 256 cores, with hardware parameters shown in Table 4-3. Relative to Chapter 3, we have increased per-core queue capacities befitting a high-performance out-of-order CPU. We keep other parameters constant on a per-core basis.

Cores	256 cores in 64 tiles (4 cores/tile), 3.5 GHz, x86-64 ISA, Haswell-like 4-wide OoO cores [82]
L1 caches	32 KB L1D & 32 KB L2I per core, 8-way, 2-cycle latency
L2 caches	1 MB, per-tile, 8-way, inclusive, 9-cycle latency
Last-level cache	256 MB, shared, static NUCA [121] (4 MB bank/tile), 16-way, inclusive, 12-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	Four 8×8 bidirectional meshes, 192-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [251])
DRAM	8 controllers, 24 GB/s each, 120-cycle minimum latency
Queues	128 task queue entries/core (32K total), 32 commit queue entries/core (8K total) 2 Kbit 8-way Bloom filters, H_3 hash functions [45]
Conflicts	Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Commit	Tiles send updates to virtual time arbiter every 100 cycles

Table 4-3: Configuration of the 256-core S5 system.

We will first examine how an application programmer can use S5 in a large real-world program before examining how S5 can be used to automatically parallelize particularly challenging graph analytics applications.

4.5.2 Achieving linear scalability on **473.astar**

First, we examine how S5 makes it easy to parallelize a complex real-world C++ program from SPEC CPU2006. As we saw in Chapter 3, the **473.astar** benchmark from SPEC CPU2006 is challenging to parallelize, and T4 was the first system to achieve speedup over serial code on **astar**. However, T4’s performance on **astar** is still limited by the frequent true data dependences involved in creating small short-lived collections stored in arrays. Thankfully, S5 provides the tools needed for C++ programmers to make small, local code changes to eliminate these data dependences. We will now describe a sequence of small code changes a programmer can apply to schedule structures in **astar**. Figure 4-12 shows the resulting performance of each parallelized implementation.

Listing 4.1 shows the hottest inner loop in **astar** in `makebound()` as well as its only callsite in the function `fill()`. These routines are used to find paths through 2D grids containing obstacles, as occur in computer games. In the inner loop, when `makebound()` visits a node indicated

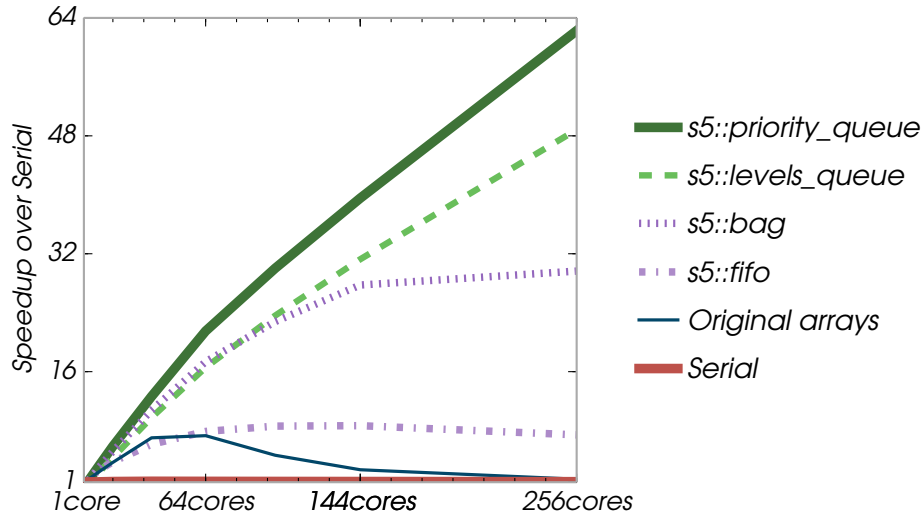


Figure 4-12: Performance comparison for six versions of *astar*, all normalized to the execution time of the serial version on one core.

by `index`, it checks the eight neighboring nodes. Each iteration of the loop performs an average of one increment of the local variable `outlen`, which together with `outp` forms a schedule structure holding nodes to be visited later. Data dependences on `outlen` cause gratuitous serialization. As a result, Figure 4-12 shows that T4 achieves only $7.3\times$ speedup on 64 cores and performance actually degrades above 64 cores, even when using the best task-splitting annotations discussed in Section 3.9.

Because the arrays pointed to by `bound1p` and `bound2p` act as queues, a programmer can simply replace these two arrays with two `s5::fifos`, as shown in Listing 4.2. `s5::fifo` guarantees nodes appear to be visited in the same order as in the original program. Figure 4-12 shows `s5::fifo` reduces pathological aborts and reaches $8.6\times$ speedup at 100 cores. This increased scalability is due to `s5::fifo::push()` not serializing all queue insertions.

Next, one may observe that the intent of this code is to perform a sort of BFS traversal on the 2D map. The global variable `step` is the distance from the source node, and each call to `for_each()` visits nodes at one distance level. The traversal terminates when no more nodes may be reached (e.g., due to obstacles) or when a target node indicated by `endindex` is reached, whichever comes first. The order in which values are visited within any single call to `for_each()` turns out not to matter: the final paths produced by the program are later determined by backtracking (not shown in the code excerpt) using only

the distances written to `waymap` during the BFS traversal. Therefore, the `s5::fifo` may be simply replaced with an `s5::bag`, a two-line change that improves `astar`'s scalability by improving load balance and reducing aborts due to data dependences on dynamic shard allocation and tracking mechanisms in `s5::fifo`. As a result, Figure 4-12 shows `s5::bag` achieves $29\times$ speedup over serial code.

So far, we have only unlocked parallelism in the inner loop, which was replaced with a `for_each()` call. Data dependences (e.g., on the frequently incremented variable `step` and on checks of whether a collection is empty) act as a serializing barrier that prevents parallelization of tasks from different invocations of `for_each()`. The queue typically holds a few hundred elements at a time, so each `for_each()` call does not contain enough work to saturate hundreds of cores for long, and serialization between `for_each()` calls limits performance on large systems.

A programmer can eliminate this bottleneck by replacing the two `s5::bags` with a single `s5::levels_queue`, as shown in Listing 4.3. This code transformation adds some overheads, chiefly due to the internal structure used by `s5::levels_queue` to ensure each between-level task runs exactly once (Section 4.4.1), but hardware-managed tasks can execute in parallel from multiple distance levels at a time. Figure 4-12 shows `s5::levels_queue` improves scalability to reach $49\times$ speedup. The improved scalability is due in part due to how `s5::levels_queue` directly passes priority values to consumer tasks, reducing memory accesses to the global variable `step`.

Finally, it turns out the final value of `step`, indicating the distance between the source and target nodes, is used in the rest of the program (not shown) only if the target node is found and `flend` is set to true. Thus, we can write to the global variable `step` less often. This actually eliminates the need for a between-levels task so the `s5::levels_queue` can be replaced with the simpler `s5::priority_queue`, as shown in Listing 4.4. This avoids the overheads imposed by `s5::levels_queue` tracking the execution of between-level events while preserving the benefits of all the previous manual optimizations. As a result, `s5::priority_queue` achieves the best performance, and Figure 4-12 shows `s5::priority_queue` achieves $62\times$ speedup on 256 cores.

These large speedups are achieved with T4 performing the vast majority of the heavy lifting by automatically parallelizing the majority

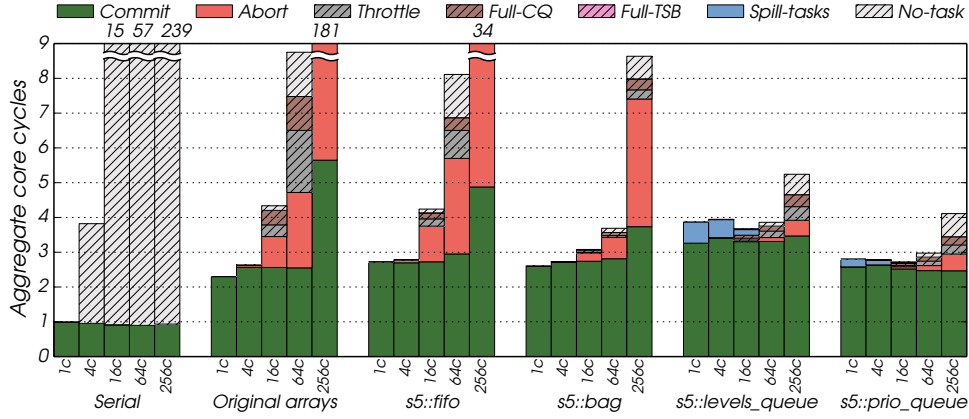


Figure 4-13: Breakdown of aggregate core cycles for six versions of *astar*, all normalized to the execution time of the serial version on one core. Bars of equal height as core counts vary indicates linear scaling.

of the thousands of lines of code in the application. S5 then enables the programmer to refactor those inner loops that presented the most difficult remaining challenge to parallelization. This refactoring was easy, chiefly it replaced some arrays with S5 ADTs and never increased the number of lines of code. The ADTs explicitly convey the scheduling semantics needed by the application, which enables the compiler and hardware to perform more effective parallelization, and even makes the intent of the code clearer to human readers.

Figure 4-13 provides further insight into how S5 changes a breakdown of how cores spend cycles. Unlike previous execution-time breakdowns such as Figure 3-11 in Chapter 3, here the total height of each bar is the product of execution time (clock cycles) and the number of cores in the system. This y-axis scaling avoids the problem of execution-time bars that shrink too small to see at high core counts, while retaining a linear scale that enables direct comparisons. The closer an implementation is to scaling linearly, the more even its bars will be in total height.

These results show that without S5, not only does T4 waste time and energy on tasks that abort, the work of merely executing the tasks that commit also increases significantly at high core counts. This is because, although T4 uses spatial hints to localize queue insertions (Section 3.9), cores still suffer cache misses at the start of each invocation of the inner loop for the ping-ponging of cache lines holding shared variables such as `outlen`. The four versions that use S5 progressively decrease

unnneeded data dependences, which reduces aborts and also improves locality which is crucial to preventing the growth of time executing work that commits on high core counts.

Examining execution time on a 1-core system shows S5 ADTs are nearly as work efficient as the arrays used in the original `astar` source code. Most of the difference in 1-core execution time between the serial code and all five parallelized versions come from T4’s use of very fine-grained tasks. `s5::levels_queue` has the highest 1-core execution time, owing to the overheads to spawn between-levels tasks and ensure the application’s between-levels callback is executed only once per level.

Ultimately only the use of `s5::priority_queue` (directly or via `s5::levels_queue`) can uncover enough parallelism to keep the aggregate time that cores spend burning dynamic power (i.e., executing tasks that commit or abort, or spilling) nearly constant across core counts. This indicates `s5::priority_queue` makes running the program on more cores not just fast but also energy-efficient.

4.5.3 Easy state-of-the-art performance for graph analytics

Now we show that these results generalize to three difficult-to-parallelize graph analytics algorithms running on a wide range of datasets. These algorithms have seen extensive prior work on software performance optimization and hardware acceleration. First, BFS is the breadth-first search we have previously described. Second, SSSP is Dijkstra’s algorithm for single-source shortest paths.⁴ BFS and SSSP are parallelized entirely automatically by UGC (Section 4.4).

Our final algorithm is KCore, which computes the k -core decomposition of a graph [208], which is a demanding stress test for a priority queue due to the large collections it stores in the queue. The output of KCore is a *coreness* value for each vertex, which can be understood as a score indicating how well-connected a vertex is to other well-connected vertices. We manually implement KCore using `s5::priority_queue` as shown in Figure 4-14. KCore begins by inserting every node in a graph into a priority queue and then gradually removes vertices while updating the priorities of remaining vertices. This pattern of iteratively reprioritizing and then removing vertices common to a broad class of

⁴ The source code for SSSP is actually the Δ -stepping algorithm, but we set $\Delta = 1$ which makes the algorithm behave exactly as Dijkstra’s algorithm, which offers both the best work efficiency and also the best overall performance on Swarm.

```

1 s5::priority_queue<int> pq;
2 for (int v : get_all_vertices()) {
3   coreness[v] = get_degree(v);
4   pq.push_init(/*priority=*/coreness[v], v);
5 }
6 pq.for_each([](uint32_t priority, int v) {
7   if (coreness[src] < priority) return;
8   for (int ngh : get_neighbors(v)) {
9     if (coreness[ngh] > priority) {
10      coreness[ngh]--;
11      pq.push(/*priority=*/coreness[ngh], ngh);
12    }
13  }
14 });

```

Figure 4-14: KCore is a popular algorithm for graph analytics which stores all vertices of a graph in a priority queue simultaneously.

Graph	Vertex count	Edge count
Road-CA [137]	1,971,281	5,533,214
road.CTR [54]	14,081,816	33,866,826
LiveJournal * [136]	4,847,571	85,702,474
Orkut * [198]	2,998,166	212,698,418

Table 4-4: Graph inputs used for evaluation. Each undirected edge is counted twice, once per direction.

difficult-to-parallelize *graph peeling* algorithms for high-quality graph clustering or decomposition. KCore and other peeling algorithms have important applications in data visualization, fraud and spam detection, analyzing documents and communities in social or web networks, and even analyzing protein-gene-disease relations in bioinformatics and medicine [140, 177, 205, 211].

We evaluate performance on real-world datasets commonly used as benchmarks in prior work, shown in Table 4-4. We selected these four datasets as inputs because they represent a wide range of variation in dataset size and graph properties such as diameter and degree distributions. Road-CA and road.CTR are road networks with weights indicating travel distances. LiveJournal and Orkut are unweighted* graphs representing friend relations in social networks. As some of these benchmarks are very difficult to parallelize efficiently on conventional CPUs, we report speedups not relative to a parallel baseline, but relative

* For SSSP, which requires a weighted graph, our datasets taken from prior work use randomly generated edge weights for these two social networks [34, 270].

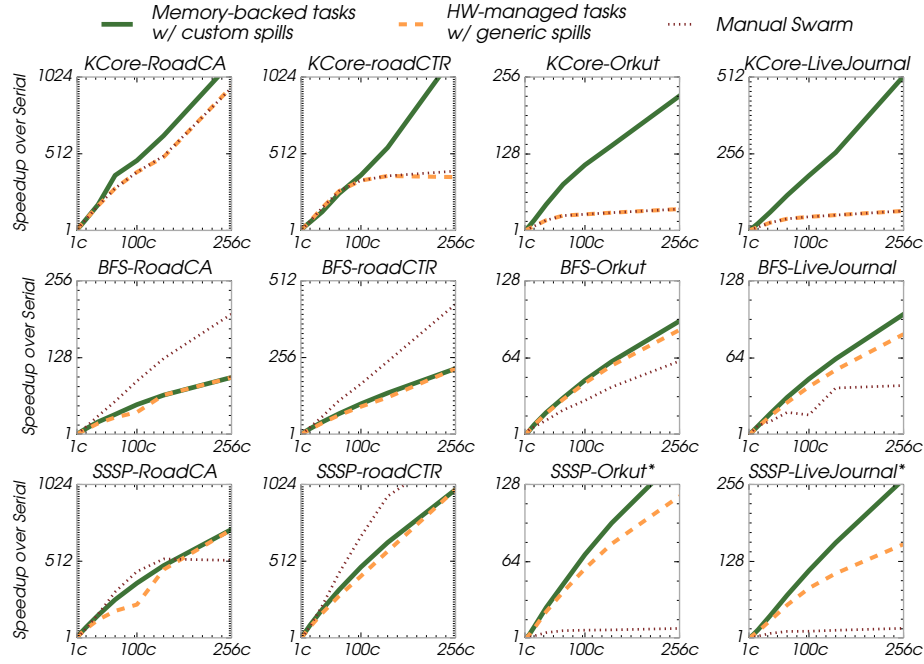


Figure 4-15: `s5::priority_queue` delivers competitive results with prior work while requiring far less programmer effort. Custom spills contribute large speedups over prior work for some applications.

to best available serial code that solves the same problem, compiled with `clang++ -O3`. The serial baseline for BFS manually manages frontiers using raw pointers to two arrays, while the serial baseline for SSSP and KCore uses the Fibonacci heap from the Boost library.

S5 vs. Prior Work: Figure 4-15 compares the performance of S5 with the best existing parallel Swarm programs. We compare these existing manually parallelized programs with two versions of `s5::priority_queue`. One version uses hardware-managed tasks and an run on the same Swarm hardware as the manually parallelized Swarm programs. This shows that S5’s easy-to-use ADTs and T4+’s automated parallelization of nested fine-grained tasks make it easy for sequential programmers to match the performance of the best available parallel code. The other version uses the memory-backed task flow enabled by S5’s new hardware, using the best of `s5::priority_queue`’s built-in custom spillers for each benchmark. Memory-backed tasks improve upon hardware-managed tasks and scale all benchmarks to 256-cores, achieving gmean 508 $\times$ and up to 1178 $\times$ speedup (on KCore-roadCTR) over serial code.

Note that the manually parallelized Swarm programs are signifi-

cantly faster than software-only multithreaded algorithms for these graph processing tasks.⁶ The manually parallelized BFS and SSSP implementations are taken from Jeffrey et al. [109], and the Swarm version of k -core is taken from Posluns et al. [183]. These prior authors performed detailed evaluations of these Swarm programs, including comparisons with software-only multithreaded CPU code [107, 109].

Large collections suffer with generic spills: Figure 4-15 shows custom spills are responsible for large performance benefits compared to both manually parallelized and S5-generated hardware-managed tasks. KCore is especially sensitive because most of the work KCore performs is in managing a huge priority queue. Figure 4-14 shows KCore initially inserts all vertices from a graph into a queue at once, and line 11 then repeatedly reinserts vertices into the queue with new priorities. Because these graph datasets have millions of vertices, this collection not only overflows hardware TQs, it often does not even fit on chip in LLC, so spilled tasks must be stored off chip in DRAM. KCore’s performance is therefore bottlenecked on limited DRAM bandwidth. When custom spillers store (priority, vertex) pairs in 8 bytes each, this results in $8\times$ less memory traffic than using generic 64-byte task descriptors, and as a result KCore-LiveJournal runs $7.8\times$ faster, as performance in this application is determined almost entirely by memory bandwidth.

Figure 4-16 presents a more detailed view showing how custom spills affect how cores spend time, both in the DRAM-bound KCore-LiveJournal but also BFS-LiveJournal, whose smaller frontiers hold subsets of vertices that often fit on chip. Storing data more compactly does result in decreased time spent handling **spills** (i.e., executing spillers and requeuers), but this is not the only effect. It also puts much less pressure on caches, so ordinary speculative tasks that **commit** obtain more cache hits and take less time as well. Another feature shown in Figure 4-16 is that, as we run the tasks on systems with increasing numbers of cores, the amount of time executing work that **commits** actually decreases and the computation becomes more efficient. This is because we carefully exploit locality, so increasing numbers of L1 and L2 caches as well as increasing LLC capacity all result in reduced cache misses on the application’s accesses to graph or vertex data. As a result, S5 scales better than linearly, explaining how Figure 4-15 shows

⁶ Note that we only consider algorithms that *exact* solutions to these problems, so we do not admit approximations for k -core decomposition that sacrifice determinism and produce lower-quality outputs [140, 205, 211].

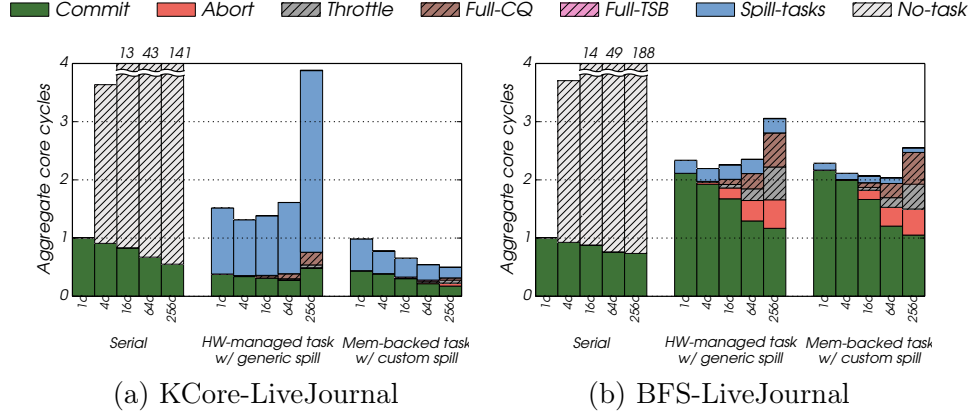


Figure 4-16: Breakdown of aggregate core cycles showing the effects of custom spillers that encode tasks compactly in memory, compared to spilling generic hardware task descriptors. Bars of decreasing height as core counts increase indicates superlinear scaling.

S5 achieves not just two but sometimes three orders of magnitude in parallel speedup on 256 cores.

4.5.4 Case study:

Custom spills for application-specific optimizations

So far, we have used custom spills to change the encoding of spilled tasks in order to reduce memory usage, but we can also use custom spills to further optimize the behavior of task scheduling. Because custom spillers are software, they can be optimized to exploit knowledge of an application’s semantics with no additional hardware change. In particular, we can have spillers identify redundant tasks and discard them from the schedule structure without executing them.

Redundant tasks may arise in an application for multiple reasons. For example, in SSSP, a task is created every time an edge is traversed, so multiple tasks are created to visit a vertex when multiple paths in the graph can reach that vertex. As SSSP computes *shortest* paths, whenever a task will perform a later visit to an already-visited vertex, the task has no effect on the result of the computation, so this redundant task is safe to drop. Similarly, in KCore, the first task to dequeue and consume a vertex from the priority queue determines the vertex’s final coreness.

To perform application-specific optimizations, application program-

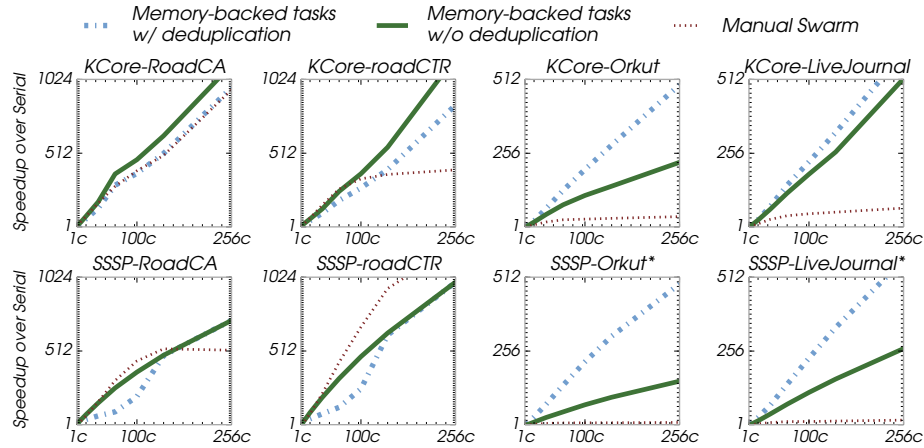


Figure 4-17: `s5::priority_queue` allows application programmers to write custom spillers. Here we evaluate the effect of custom spillers that drop redundant tasks.

mers extend `s5::priority_queue` with custom spillers for deduplication. `s5::priority_queue` takes optional custom spillers as a template parameter, which it will register in place of its built-in spillers. We pass `s5::priority_queue` a custom spiller that uses untracked memory to store the lowest timestamp that has been seen for each element value in the collection (e.g., for each vertex in the graph). Our new custom spillers and the custom requeuers it spawns compare each task’s timestamp with the previously seen timestamp, and if they encounter a task with strictly higher timestamp (worse priority) than a previously spilled task that consumes the same element value, the higher-timestamp task is simply dropped.

Figure 4-17 shows our deduplication has highly variable performance impacts and sometimes achieves significant benefits. Since deduplication is controlled entirely by application programmers, they could use it only when their application benefits, or even implement alternative deduplication strategies specific to their applications. S5’s custom spillers never force users to pay for a feature they do not need.

Deduplication more than doubles performance for some social network graphs which have many edges per vertex (which give redundant paths to a vertex in SSSP and repeated decrements of priorities in KCore). For example, custom spills with deduplication makes KCore-Orkut run $13.5\times$ faster than the best available parallel Swarm program and $515\times$ faster than the best available serial code. The form of deduplication we implemented does add complexity and execution time

overheads to custom spillers and requeuers. In road graphs where there are relatively few redundant visits to any given vertex, the costs outweigh the benefits so a user could simply choose not to use the deduplicating custom spiller.

4.6 Additional Related Work

4.6.1 Hive

Posluns et al. recently proposed an extension to the Swarm architecture called Hive [183], which introduces new hardware mechanisms to provide strict guarantees that redundant tasks are deduplicated. Whereas S5 only opportunistically deduplicates spilled tasks, Hive uses intricate logic to track speculatively redundant tasks within a hardware TQ, correctly handling when a task becomes redundant and then non-redundant if some other task aborts. In applications where few tasks spill from TQs, Hive is able to perform better deduplication at the cost of invasive changes to the TQ implementation. The biggest cost of Hive is the energy and area expense of new associative hardware structures that perform lookups upon every enqueue to determine which tasks have become redundant within a hardware TQ.

In return for this hardware complexity, Hive’s strong deduplication guarantees simplify the job of software. For example, line 7 in Figure 4-14—which performs a software check to ensure redundant tasks exit with no effect—is unneeded in the Hive version of KCore, because Hive eliminates all redundant tasks. Hive also changes the format of hardware task descriptors to include unique task IDs which help identify redundant tasks with the same timestamp. By contrast, when an S5 custom spiller repeatedly sees a task with the same priority and arguments, the spiller cannot as easily identify whether one task is being spilled again after being requeued, or if there are redundant tasks and one can be dropped. Future work could investigate combining S5’s custom spills with Hive’s unique task IDs.

Despite using simpler hardware than Hive, S5 can achieve better speedups than Hive thanks to its flexibility. We use this flexibility to combine deduplication with compact application-specific task encodings. As a result, S5 obtains $13.5\times$ speedup over Swarm on KCore-Orkut, which is the benchmark on which Hive reported its best speedup: a

mere $2.8\times$ improvement over Swarm due to using deduplication alone. S5’s speedups were also achieved with ADTs that are easier to use than Swarm’s and Hive’s low-level task programming interfaces. Moreover, S5 reuses Swarm’s original hardware priority queue design as-is and makes only simple changes to the small hardware spill buffer, so the speedups of memory-backed tasks are a nearly free lunch. By contrast, Hive dedicates silicon to track redundant tasks, incurring an area opportunity cost even for applications that do not benefit from deduplication. Finally, S5 is flexible and programmable because it gives software control over adapting scheduling behavior rather than implementing new behaviors it in hardware. This means, after S5 is deployed, future programmers may continue to extend S5’s memory-backed tasks with data structure optimizations and behaviors that benefit new applications.

4.6.2 Concurrent data structures

In the context of explicitly multithreaded programs, extensive work has shown concurrent data structures can reduce serialization [91, 155]. But S5 is the first library of data structures to eliminate serialization while maintaining sequential semantics.

A common means of making high-quality data structures easy to adopt is to package them as a modular library of generic data types. General-purpose programming languages such as C# and Java now incorporate generic concurrent collections such as scalable bags and dictionaries in their standard libraries [133, 152, 153]. S5 takes inspiration from how these prior approaches hide implementation complexity behind simple abstract interfaces, which is reflected in the interface of S5’s ADTs. But while libraries for concurrent data structures can hide data layout and encoding details, they do not hide the inherent difficulty of concurrent programming. Using these libraries still requires programmers to manually reason about which tasks are independent and which pieces of application logic needs additional synchronization, demanding careful thought to avoid deadlock and race conditions [4, 76, 141]. By contrast, S5 preserves strong sequential semantics, making it impossible to produce results that cannot be produced by a serial single-core execution. As a result, using S5 is as easy sequential programming.

Researchers continue to propose new specialized domain-specific data structures and algorithms. Because S5 allows software to store spilled tasks in an arbitrary backing store, ongoing research could

be incorporated into future extensions of S5. For example, a significant amount of effort has gone into software frameworks that perform specialized scheduling structure optimizations for graph processing [56, 74, 80, 95, 135, 178, 212, 213, 228, 229]. Future work can extend S5 to provide these optimization choices for memory-backed tasks that could then be selected by a programmer or chosen automatically by the GraphIt compiler.

While the implementation of some parallel data structures can be accomplished in existing parallel systems, S5 builds on T4’s nested speculative parallelization to unlock more parallelism in more contexts than is possible in prior work. By contrast, state-of-the-art parallel frameworks for both general and domain-specific computation restrict the actions allowed in each task (e.g., forbidding arbitrary writes to shared data) and make extensive use of bulk-synchronous parallelism (i.e., running one parallel loop or kernel to completion before launching the next) that does not exploit parallelism between different invocations of parallel kernels [96].

4.6.3 Hardware support for task scheduling

There is extensive prior work on hardware support for general communication primitives that are flexible enough to support many software patterns. At a bare minimum, a shared-memory system with support for atomic operations can be used to implement parallel software runtimes [118]. Some systems provide hardware support for user-level interrupts to trigger handlers that move data payloads through shared memory [33, 159, 219, 254]. Direct hardware support for point-to-point message passing makes some communication patterns faster [5, 164, 220]. Active messages [249] improves upon message passing by allowing each message to specify an interrupt handler that should be executed upon delivery. UDM [145] improved upon active messages with greater flexibility for software to choose dynamically between asynchronous (interrupt-based) delivery or queuing messages until fetched by a recipient. ADM [204] is similar to UDM, but allows a fast user-level interrupts to cooperate efficiently with occasional OS kernel scheduling, and is efficiently implemented using register-to-register communication within a shared-memory chip multiprocessor that can support more general programs. All of these message-passing primitives may be used by a software runtime system that must still store application tasks

using separate in-memory structures. All of these systems also only served explicitly parallel programs. By contrast, S5 performs implicit and mostly automatic parallelization while allowing programmers to exploit hardware queues to schedule deferred tasks.

Extensive prior work has also proposed fixed-function hardware to accelerate narrow forms of task scheduling. Carbon [128] provides hardware support for fork-join parallel programs, while several other prior systems provide systems to accelerate barriers in bulk-synchronous programs [23, 168, 207]. Streaming multicores [50, 52, 119, 237] use FIFOs to facilitate computations that consume streams of data in the order they are produced. Dataflow architectures (Section 2.1) facilitate delivering multiple messages to one recipient, the last of which to arrive triggers the execution of the recipient task. There are even relaxed-priority scheduling architectures, which dispatches tasks to cores using dedicated hardware priority queues [49] or with specialized hardware handling in-memory priority queues [88]. Each of these techniques benefits a different style of parallel programs. A general dataflow program cannot safely and efficiently run on a relaxed priority scheduling architecture, nor can a general fork-join parallel program be trivially recompiled for a streaming multicore. And no prior technique before Swarm can be used to parallelize an application that needs strict priority ordering.

By contrast, sequential order is very general because nearly all computations are simpler to express sequentially than as an explicitly parallel program. Thus, an architecture that allows software to specify an arbitrary sequential order in which tasks should execute is very general. Swarm is that architecture, and S5 provides a new interface to Swarm so sequential applications can achieve easy, effective, and efficient parallelization with nearly arbitrary ordering requirements. While Swarm’s distributed mechanisms resemble those in relaxed-priority architectures, Swarm’s mechanisms guarantee strict global ordering that these other systems did not provide. We leverage Swarm’s strictly ordered semantics to make S5 far more general and easier to use than prior systems.

4.6.4 Compilers using application knowledge to reduce dependences

In general, compilers seek to exploit facts they can prove by analyzing source code in order to generate fast code that uses hardware well.

Parallelizing compilers attempt to determine when data dependences are unneeded and can be eliminated through privatization (Section 2.2). However, in many cases effective parallelization requires understanding the programmer’s high-level intent in order to determine how a program can be transformed to eliminate those implementation details that cause serialization while preserving its desired behavior.

Within speculative parallelization, some prior work proposes compilers or programming frameworks that exploit semantic commutativity with programmer support [37, 53, 127], which requires programmers to annotate certain tasks or code regions (e.g., loop iterations) as being commutative (i.e., valid to execute in any order). Different invocations of these commutative tasks make use of privatized data, that is, local copies of temporary or intermediate scalar values. But none of these systems allows the system to entirely skip storing data in complex in-memory structures. And these systems lacked parallel composability, so they did not parallelize entire programs by exploiting nested parallelism across invocations of parallelized kernels to keep hundreds of cores busy.

4.7 Summary

S5 reuses Swarm’s distributed hardware priority queues and strict priority semantics as a general scheduling primitive that replaces in-memory schedule structures. This is possible due to the generality of strict priority ordering, which can easily implement many sequential scheduling patterns. Despite its generality, Swarm’s hardware queues outperform specialized software-only schedule structures constructed in shared memory. Hardware priority queues can be so fast because they are implemented with dedicated, fully pipelined hardware mechanisms that are placed near cores in a large multicore system. S5 extends this hardware to enable a memory-backed task flow that combines the speed and parallelism of Swarm’s hardware queues with the capacity efficiency and specialized semantics of application-specific in-memory structures. As a result, S5 achieves gmean $508\times$ speedup over serial code while maintaining the ease of writing sequential programs.

```

1 void checkNeighbor(int index1, int* outp, int* outlen) {
2   if (waymap[index1].fillnum != fillnum && maparp[index1] == 0) {
3     outp[(*outlen)++] = n;
4     waymap[index1].fillnum = fillnum;
5     waymap[index1].num = step;
6     if (index1 == endindex) {
7       flend = true;
8     }
9   }
10 }
11 int makebound(int* inp, int inlen, int* outp) {
12   int outlen = 0;
13   for (i = 0; i < inlen; i++) {
14     index = inp[i];
15     checkNeighbor(index-yoffset-1, outp, &outlen);
16     checkNeighbor(index-yoffset, outp, &outlen);
17     checkNeighbor(index-yoffset+1, outp, &outlen);
18     checkNeighbor(index-1, outp, &outlen);
19     checkNeighbor(index+1, outp, &outlen);
20     checkNeighbor(index+yoffset-1, outp, &outlen);
21     checkNeighbor(index+yoffset, outp, &outlen);
22     checkNeighbor(index+yoffset+1, outp, &outlen);
23   }
24   return outlen;
25 }
26 bool fill(...) {
27   // bound1p and bound2p point to arrays
28   ...
29   bound1p[0] = startidx;
30   int bound1len = 1;
31   // fillnum, flend, endindex, and step are global variables
32   fillnum++;
33   flend = false;
34   endindex = ...;
35   step = 1;
36   while (true) {
37     int bound2len = makebound(bound1p, bound1len, bound2p);
38     step++;
39     if ((bound2len == 0) || flend) break;
40     bound1len = makebound(bound2p, bound2len, bound1p);
41     step++;
42     if ((bound1len == 0) || flend) break;
43   }
44   ...
45 }

```

Listing 4.1: Serial code taken from Way_.cpp in *astar*, compressed and simplified for clarity.

```

1 void checkNeighbor(int index1, s5::fifo* out) {
2   if (waymap[index1].fillnum != fillnum && maparp[index1] == 0) {
3     out->push(n);
4     waymap[index1].fillnum = fillnum;
5     waymap[index1].num = step;
6     if (index1 == endindex) {
7       flend = true;
8     }
9   }
10 }
11 bool makebound(s5::fifo* in, s5::fifo *out) {
12   in->for_each([out](int index) {
13     checkNeighbor(index-yoffset-1, out); // NW neighbor
14     checkNeighbor(index-yoffset, out); // N neighbor
15     ...
16   })
17   return out->empty();
18 }
19 bool fill(...) {
20   s5::fifo queue1, queue2;
21   queue1.push(startidx);
22   // fillnum, flend, endindex, and step are global variables
23   fillnum++;
24   flend = false;
25   endindex = ...;
26   step = 1;
27   while (true) {
28     bool empty = makebound(&queue1, &queue2);
29     step++;
30     if (empty || flend) break;
31     empty = makebound(&queue2, &queue1);
32     step++;
33     if (empty || flend) break;
34   }
35   ...
36 }

```

Listing 4.2: astar using s5::fifo

```

1 static s5::levels_queue queue;
2
3 void checkNeighbor(int index1, int priority) {
4     if (waymap[index1].fillnum != fillnum && maparp[index1] == 0) {
5         queue.push(n);
6         waymap[index1].fillnum = fillnum;
7         waymap[index1].num = priority;
8         if (index1 == endindex) {
9             flend = true;
10        }
11    }
12}
13 return out->empty();
14}
15 bool fill(...) {
16     queue.push_init(1, startidx);
17     // fillnum, flend, endindex, and step are global variables
18     fillnum++;
19     flend = false;
20     endindex = ...;
21     step = 1;
22     queue.for_each([](uint64_t priority, int index) {
23         checkNeighbor(index-yoffset-1, priority); // NW neighbor
24         checkNeighbor(index-yoffset, priority); // N neighbor
25         ...
26     }, [](uint64_t priority) {
27         step++;
28     });
29     ...
30 }

```

Listing 4.3: astar using s5::levels_queue

```

1 static s5::priority_queue queue;
2
3 void checkNeighbor(int index1, int priority) {
4     if (waymap[index1].fillnum != fillnum && maparp[index1] == 0) {
5         queue.push(n);
6         waymap[index1].fillnum = fillnum;
7         waymap[index1].num = priority;
8         if (index1 == endindex) {
9             flend = true;
10            step = priority + 1;
11        }
12    }
13 }
14 return out->empty();
15 }
16 bool fill(...) {
17     queue.push_init(1, startidx);
18     // fillnum, flend, endindex, and step are global variables
19     fillnum++;
20     flend = false;
21     endindex = ...;
22     step = 1;
23     queue.for_each([](uint64_t priority, int index) {
24         checkNeighbor(index-yoffset-1, priority); // NW neighbor
25         checkNeighbor(index-yoffset, priority); // N neighbor
26         ...
27     });
28     ...
29 }

```

Listing 4.4: astar using s5::priority_queue

5 ASH: Accelerators for Simulating Hardware

This work was conducted in collaboration with Fares Elsabbagh, Shabnam Sheikha, Quan Nguyen, Joel Emer, and Daniel Sanchez. The individual contributions of this dissertation are highlighted in Section 5.1.1.

5.1 Introduction

Specialized hardware accelerators are increasingly common due to large performance benefits over general-purpose processors. Specializing hardware for a specific application eliminates the overheads of features the application does not need and makes hardware simpler. Simpler hardware elements can improve performance because they are easy to replicate many times on a chip to exploit parallelism and easy to tightly integrate, improving communication costs. But making hardware simpler may sometimes require forgoing powerful abstractions previous chapters have relied on, such as a coherent cache hierarchy, leaving software to use more explicit communication and synchronization primitives.

In this chapter, we demonstrate a specialized accelerator and compiler can handle explicit communication and can still benefit from the same principles of fine-grained parallelization used in Swarm and T4. We specialize for *Register-Transfer-Level (RTL)* simulation, i.e., the precise simulation of a hardware design written in a Hardware Description Language (HDL) like Verilog. RTL simulation is the bottleneck in computer chip design, as it is needed to evaluate how design improvements trade off performance, power, and area; to verify the design’s correctness; and to debug implementation errors. In particular, verification dominates chip design cost [100].

Unfortunately, existing simulators are slow. The fastest software

simulators, like Verilator [215], are purpose-built optimizing compilers that translate RTL code into an efficient software program (e.g., in C++) [22, 181, 215]. But simulators scale poorly across CPU cores, because parallelism is fine-grained: work must be split into tiny tasks with few operations each. Additionally, these tasks have many data dependencies and require frequent synchronization. As we will see (Section 5.2), conventional multicores incur excessive synchronization overheads that erase most of the benefits of parallelization. Without effective parallelization, slow simulation delays the design of new hardware, limits the efficiency of the resulting design (due to insufficient design exploration), and makes hardware bugs more likely (due to insufficient verification coverage).

Since simulators are so slow, chip designers often resort to specialized *emulators*, which map a design to hundreds to thousands of FPGAs [42, 232] or specialized processors [24, 41]. Emulators work at a lower level of abstraction (gates), which limits their efficiency; are large and expensive; and suffer very long compile times, *days to weeks* for large designs [99]. This limits emulators to final integration testing, when the design is nearly finished and bug-free. Simulators compile designs in minutes, so they are the far better option for most of the design cycle, when changes are frequent (Section 5.2.4). Thus, we focus on accelerating software simulation, retaining its fast compilation while achieving dramatic speedups.

We propose the ASH (Accelerator of Simulated Hardware) system, a carefully co-designed architecture and compiler for RTL simulation. ASH reuses simple a minimal set of key techniques from Swarm and T4 and adds a new dataflow synchronization primitive needed to enable efficient execution of RTL simulation. We present two ASH variants that target the key acceleration opportunities in RTL simulation:

Dataflow execution with DASH: RTL simulation is naturally expressed as many tasks whose outputs drive the inputs of other tasks, so it is a perfect match for dataflow execution (Section 2.1). Dataflow execution has the potential to expose abundant parallelism, as each task can run as soon as its inputs have been produced. Unfortunately, RTL simulation fails to scale on current multicores because it needs small tasks to expose sufficient parallelism, but small tasks have prohibitive overheads that negate the benefits of parallelism.

DASH (Dataflow ASH) tackles this challenge by providing novel hardware mechanisms for dataflow execution of short tasks. We extend

a multicore with hardware to orchestrate dataflow execution: DASH hardware gathers inputs and dispatches tasks to cores after inputs are available. DASH implements a novel *prioritized* dataflow execution that orders tasks to reduce in-flight tasks, avoiding the overheads of prior dataflow architectures [64, 163, 230]. DASH gathers inputs through register-to-register communication that moves data between cores, allowing DASH to have a simple, partitioned memory system without cache coherence.

Selective event-driven execution with SASH: In most digital designs, only a small fraction of the signals change each cycle [21, 62, 122], so simulating the whole design every cycle is wasteful. *Selective execution*—simulating *only* the active fraction of the design—is more efficient. Prior work has proposed *event-driven* simulators and architectures [2, 47, 68, 70, 104] where work happens through events that may schedule other events for later execution. Unfortunately, these prior systems lack support for dataflow execution, so running dataflow tasks selectively incurs costly synchronization through memory that negates the benefits of selective execution.

SASH (Selective event-driven ASH) tackles this challenge by extending DASH with selective execution, running *only* tasks whose inputs change during a given cycle. Selective execution introduces *dynamic* data dependences: each task receives inputs only from producers that have run, but not from skipped ones. SASH handles these dependences with *speculative execution*, running each task with its received inputs and using old values for non-received ones (i.e., speculating that they aren’t changing this cycle). A late-received input causes the task to abort and re-execute. Whereas Swarm tasks speculate because the set of tasks and their potential data dependences is unknown, SASH tasks speculate for a different reason: although all dependences are statically known to the compiler, determining when each task is safe with certainty would require extra work and communication, and it is more efficient to speculate.

ASH compiler: The ASH compiler builds on Verilator and introduces novel techniques to automatically parallelize RTL efficiently, place tasks and data to minimize communication. To make effective use of DASH’s hardware management of priority-ordered tasks as well as SASH’s speculative execution, the ASH compiler exploits many of the same principles as T4 and S5, using trees of tiny tasks to maximize parallelism and

limit abort costs as well as assigning tasks timestamps to encode their priority order.

In summary, ASH leverages classic compiler and architecture concepts including speculative and dataflow execution (Section 2.1), and contributes new techniques to make their combination efficient and to use them automatically. While prior systems used either speculation [2, 69, 107] or task-level dataflow [64, 256], ASH is the first to combine them, achieving good scalability and avoiding ineffectual work.

We evaluate ASH in simulation, on four large Verilog designs that include CPU cores, GPU cores, and accelerators. Our contributions make RTL simulation scale to hundreds of cores. A 256-core ASH system is gmean $1,485\times$ faster than serial Verilator on a system with a single (simple) core, and gmean $32\times$ faster than parallel Verilator on a commercial CPU with 32 complex cores while using $3\times$ less area.

5.1.1 Contributions

ASH’s execution models, compiler, and hardware implementation were developed collaboratively. This chapter describes the entire design of ASH, while emphasizing the author’s individual contributions, which we summarize here.

ASH compiler support for tiny tasks: The ASH compiler adopts techniques from T4, such as the generation of trees of spawner tasks that exploit distributed hardware task units to produce argument descriptors in parallel. In ASH, spawner trees perform efficient multicast messaging (Section 5.4.3.5). Similar to T4, the ASH compiler automatically generates timestamps as well, but uses a different algorithm that prioritizes tasks using information from the statically known dataflow graph (Section 5.4.3.4). Finally, we contribute new compiler techniques to coarsen tasks to reduce task creation and argument-passing overheads while still limiting the size of the resulting tasks (Section 5.4.3.3).

ASH hardware implementation: ASH’s hardware structures must handle the need to merge multiple argument descriptors into a single task before DASH can safely execute it or SASH can safely commit it. ASH hardware exploits prioritization with unique timestamps to ensure matching argument descriptors can be found within ASH’s finite-size AQ & merge-unit, even when Aqs overflow (Section 5.4.2). SASH’s commit protocol also differs from Swarm’s to ensure tasks to not commit when they have a late-arriving descriptor in flight in the NoC (Section 5.5.2).

Additionally, a hardware heuristic was added that throttles the dispatch of speculative SASH tasks to reduce aborts (Section 5.8).

Evaluation: The baseline Verilator system generates serial or multithreaded C++ code, which is compared against ASH (Section 5.9). Several effects of varying task granularity and coarsening were evaluated and analyzed, showing the importance of hardware support for short tasks (Section 5.2.2 and Section 5.2.3). To evaluate ASH, the Swarm simulator infrastructure was extended to model ASH hardware, with support for partitioning and mapping tasks and data (Section 5.8). The occupancy of SASH’s hardware queues was evaluated, showing how SASH exploits out-of-order execution within a finite speculation window (Section 5.9.3). The same simulator infrastructure was also used by collaborators for other parts of the ASH evaluation.

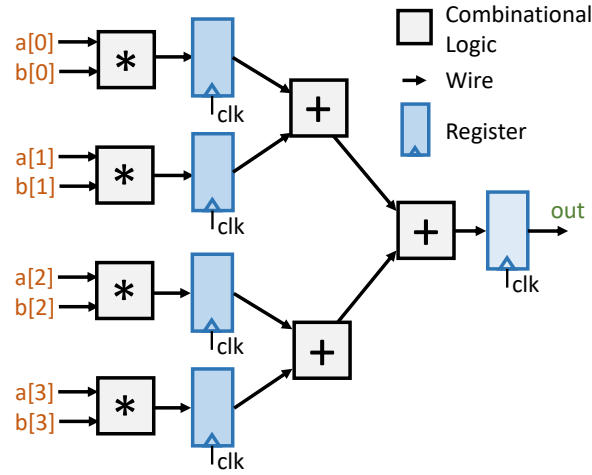
5.2 Motivation and Background

In this section, we first introduce the key characteristics of RTL simulation. We then present its two key optimization opportunities and quantify the limitations of existing systems, including parallel simulators and FPGA emulators.

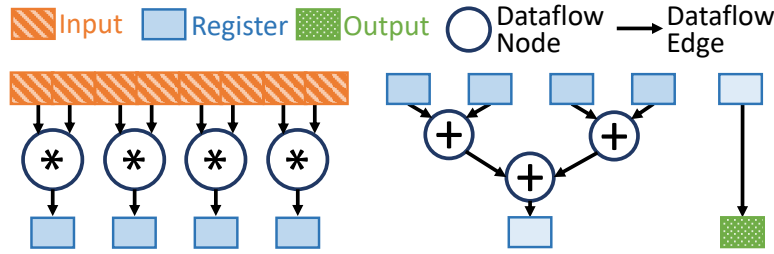
5.2.1 Understanding RTL simulation

RTL simulation functionally evaluates a digital circuit written in an HDL like Verilog. RTL simulation precisely evaluates how signals and state elements (e.g., registers) evolve over clock cycles. For instance, Figure 5-1(a) shows a 2-stage pipeline that computes the dot-product of two 4-element vectors. In an RTL simulation, a testbench feeds inputs each simulated cycle, and compares the outputs (and maybe other signals) against a reference model. It may also evaluate performance (e.g., cycles taken on a long computation) and power (by measuring component activities). Thus, RTL simulation verifies digital designs and analyzes their efficiency.

In this work, we focus on simulating *synchronous* circuits, by far the dominant design style (other styles, like asynchronous circuits [39], are rarely used). These systems comprise combinational logic and clocked registers. (For simplicity, we assume that all registers are driven by one clock, but our techniques also work with multiple clock domains [243].)



(a) Example 2-stage pipeline.



(b) Dataflow graph for (a).

Figure 5-1: RTL simulation of synchronous circuits can be abstracted as a dataflow graph.

Dataflow graphs: RTL simulation of synchronous circuits can be abstracted as the execution of a *dataflow graph*, where nodes represent combinational logic and edges represent data values. Figure 5-1(b) shows the dataflow graph for the circuit in Figure 5-1(a), with **inputs** on the top and the only **output** on the bottom. The same **registers** are shown on both top and bottom, as they are read and written every cycle. RTL simulation evaluates each node in the dataflow graph once per simulated cycle in an order consistent with its dataflow edges.

RTL simulators: Modern RTL simulators like Verilator [215], ESSENT [22], RepCut [250], and Cuttlesim [181] are *compilation-based*: they translate HDL code to a dataflow intermediate representation (IR), then translate the IR to a software program that simulates the circuit. For example, Figure 5-2 shows how an ALU is translated from Verilog to efficient C++. Translation leverages the semantic similarities of

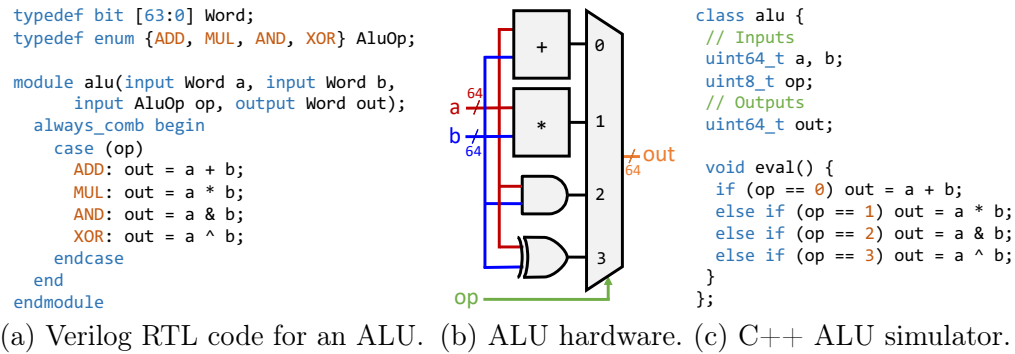


Figure 5-2: Fast RTL simulators compile HDL code (e.g., Verilog) to a software program that simulates the hardware.

HDLs and software languages, e.g., arithmetic operations are directly translated to C++ arithmetic operations, not bit-level operations.

RTL simulation admits two key optimizations: executing the dataflow graph in parallel, and executing only the nodes whose inputs change. We now quantify how existing systems with inadequate hardware support fall short.

5.2.2 Parallel execution needs small tasks

RTL simulators can parallelize execution by leveraging the dataflow graph, which exposes available parallelism: independent paths can be evaluated in parallel. However, fine-grained dependences cause serialization: for instance, in Figure 5-1(b) the adder tree can be partially parallelized, but two inputs feed into the final adder. In practice, combinational paths have tens of nodes, with edges fanning in and out, creating complex dependences that make parallel execution challenging.

The granularity of scheduling is a key aspect of parallelization. Each *node* in the dataflow graph performs a tiny amount of work, often just a single assignment. On current multicore systems, communication overheads would make individually scheduling these nodes extremely expensive. Instead, dataflow nodes are first *merged* into larger *tasks*. Then, this *task-level dataflow graph* is scheduled across cores.

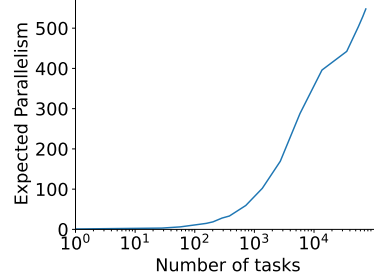
Merging nodes into larger tasks introduces delicate tradeoffs between work-efficiency and parallelism. On the one hand, larger tasks are more work-efficient. On the other hand, merging too much has two ill effects: it *reduces parallelism* and *creates tasks with many inputs and*

outputs, which are harder to schedule. For instance, merging the four independent multiplier nodes in Figure 5-1(b) reduces parallelism by $4\times$ and produces a single task with 8 inputs and 4 outputs.

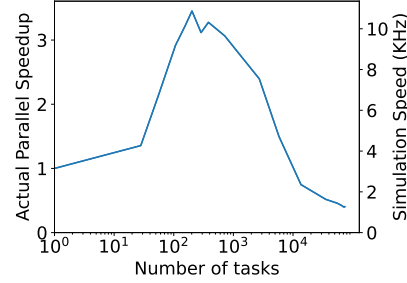
The problem is that existing multicores force simulators to create huge sequential tasks that sacrifice most of the parallelism. This happens because communication and synchronization are very expensive: communicating a single value across cores is done through shared memory, and costs hundreds of cycles. Tasks must have thousands of operations to amortize these costs. But larger tasks also have more dependencies, inducing needless serialization that tanks parallelism.

To quantify this problem, we use Verilator [215] to compile and simulate Chronos, a graph-processing accelerator with 128 processing engines (see Section 5.8 for methodology details). Verilator iteratively merges dataflow nodes into large tasks. It tries to avoid lengthening the *critical path*, i.e., the costliest chain of tasks from initial values (inputs or registers) to final values (outputs or registers). Merging nodes can produce a costlier task on the critical path, so Verilator stops merging when the *parallelism*, i.e., the ratio of total cost of all tasks to critical path cost, reaches a threshold.

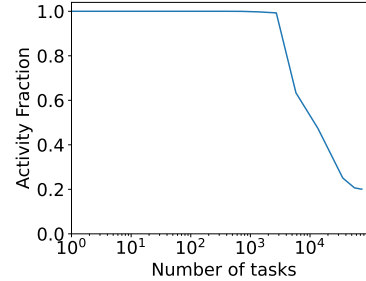
Figure 5-3(a) shows how parallelism grows with the number of tasks. The x-axis uses a logarithmic scale to show a wide range of options: from serial simulation using one task, to restricting Verilator to only do merges that avoid hurting the critical path, which produces 74 K tiny



(a) Finer tasks increase parallelism.



(b) Actual measured performance.



(c) Tiny tasks uncover low activity.

Figure 5-3: Effects of varying task size for Verilator. More tasks (x-axis) means finer grain.

tasks. Figure 5-3(a) shows that parallelism is plentiful, but only with many small tasks: merely achieving $100\times$ parallelism requires about 2000 tasks.

Figure 5-3(b) shows how this potential parallelism translates to performance. It reports speedup over serial Verilator across the range of task counts (and granularities) on a 32-core AMD Zen 2 CPU. Verilator schedules tasks statically to a fixed number of threads (a good strategy, as task costs are known at compile time). We adapt Verilator to allow varying threads and merging level independently, and report the performance of the best thread count for each merging level.

Unfortunately, Figure 5-3(b) shows that parallel execution yields limited speedups: the best performance is for 203 tasks, which is only $3.5\times$ faster than serial simulation despite an expected parallelism of 18.7. Such limited speedups are typical for Verilator’s multithreaded simulations [216, 217]. This is because, even with 203 relatively large tasks, each task has only about a microsecond worth of work, and synchronization costs dominate. These costs make speedups plummet with smaller tasks. DASH’s support for dataflow execution avoids these costs, enabling small tasks with high parallelism.

5.2.3 Selective execution needs small tasks

Digital systems have low activity factors: a small fraction of the logic switches each cycle [21, 62, 122]. Simulators can exploit this by evaluating only the nodes whose inputs change.

Effective *selective execution* demands small tasks. Figure 5-3(c) reports the fraction of work performed by *active* tasks, averaged across simulated cycles. A task is active and must execute if any of its inputs has changed since it last executed. With small tasks, active tasks make up only 20% of work. But larger tasks make it more likely that some logic within each task is active, making inactive tasks rare. Below 2000 tasks, the activity factor is pegged to 100%.

Prior work has developed serial simulators that exploit low activity factors through conditional execution [22]. But the inability of parallel simulators to deal with small tasks makes selective execution ineffective, and to our knowledge, state-of-the-art parallel RTL simulators are non-selective [216].

In parallel simulation, skipping inactive tasks adds complexity because any active task must now know which inputs change and which

stay the same. Prior work has proposed *conservative* and *optimistic* parallelization approaches. With conservative parallelization, as done in the Chandy-Misra-Bryant (CMB) algorithm [47], each inactive task must send *null messages* to indicate lack of changes in its outputs. This approach still traverses the full dataflow graph each cycle—it skips only the work within each task, *but not the communication*. For the tiny tasks needed in RTL simulation, this is inefficient.

Time Warp [104, 105] avoids null messages and conservative synchronization using an optimistic approach: tasks execute speculatively, assuming that no other inputs will be received, and an input received out of order causes the task to abort and re-execute. But Time Warp has no concept of dataflow execution: each task carries one argument from a single simulated object to another. Thus, while prior accelerators used this optimistic approach in other domains [2, 68, 107], they lack the hardware support needed to manage speculative execution of *dataflow tasks*. As we later quantify (Section 5.10), ASH is the first system with sufficient hardware support to make selective execution profitable in parallel RTL simulation.

5.2.4 Emulation versus simulation

Hardware emulators are the only platforms capable of running billion-gate designs at MHz speeds by mapping gates directly to hardware elements implemented on FPGAs or specialized gate processors. However, their long compile times make them incompatible with crucial parts of the chip design process, such as design space exploration and prototyping, when RTL changes are extremely frequent.

Current FPGA-based emulators include Cadence’s Protium [42] and Synopsys’s ZeBu [232]. Their compilation times span from days to weeks, as partitioning, placing, and routing a design across many FPGAs is a very complex process. Processor-based emulators use a large collection of ASICs instead of off-the-shelf FPGAs: Cadence’s Palladium [41] uses a massive array of Boolean processors; and Mentor’s Veloce Strato [151] uses custom FPGAs tailored to emulation. (Section 5.10.3 describes additional systems.) ASIC-based systems have lower compilation times (hours to days) and better debugging capabilities than FPGA-based ones, but they are slower and more expensive. Thus, emulators are restricted to late in the design process, when changes are rare.

Simulation is the best choice for evaluating RTL designs at high

System	Compile time	Sim speed	Time to simulate		
			1M cycles	1B cycles	1T cycles
SW sim.	2 mins	11 KHz	3.9 mins	1.1 days	3 years
SASH	2 mins	414 KHz	2.4 mins	43 mins	28 days
FPGA $\times 2$	13 hours	1.4 MHz	13.2 hours	13.4 hours	8.7 days
SASH/FPGA			332 $\times$	18.9 $\times$	0.3 $\times$

Table 5-1: Simulation vs. FPGA emulation.

speeds and at large scales. To concretely quantify its advantages, Table 5-1 shows the times to compile, and then run, Chronos in (1) pure software simulation, (2) SASH, and (3) emulation on 2 FPGAs. Simulated duration is important, because different activities demand different durations: one million cycles ($\sim 1ms$) suffices for short tests, one billion cycles ($\sim 1s$) enables thorough performance evaluation, and one trillion cycles (~ 20 min) enables running complex software.

To benchmark emulation, we use a system with two large Alveo U250 FPGAs [255] directly connected through 200Gbps links. Chronos does not fit in one FPGA. By manually partitioning the design and applying Virtual Wires [18], we achieve a 1.4MHz speed, limited by inter-FPGA latency. After days of tuning, this design compiled in *13 hours* (running both FPGA compiles in parallel). Commercial emulators can automate partitioning but at the cost of longer compile times.

SASH has a somewhat lower simulation speed than emulation, but compiles Chronos in *2 minutes*. This enables a much shorter design and debug cycle than emulation: in the time it takes to carry out a billion-cycle FPGA emulation run, engineers could have done nearly 20 SASH runs.

5.3 System Overview

Figure 5-4 shows an overview of the ASH architecture. ASH consists of a co-designed hardware architecture and RTL compiler. The architecture efficiently supports the two key optimizations identified in Section 5.2: it scales simulation to many cores using dataflow execution, and it avoids needless work through selective execution. The RTL compiler leverages these architectural features to achieve high performance.

ASH hardware is designed from the ground up to support fine-grain tasks. The ASH chip is organized similarly to Swarm, with

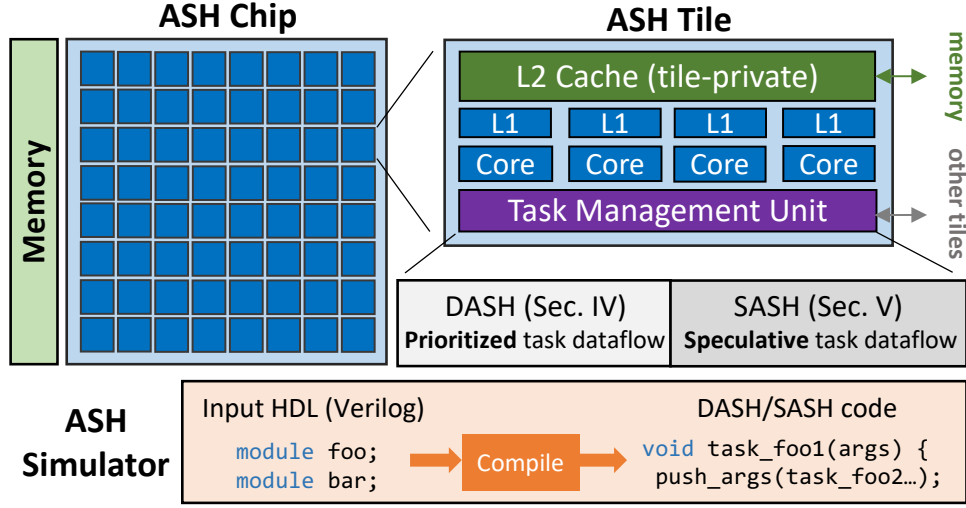


Figure 5-4: ASH system overview.

tiles that each have four cores and a *task unit*. However, the task units are changed to support the specific needs of dataflow simulation workloads. We design two variants of ASH, and present them in stages: DASH (Section 5.4) supports task dataflow specialized to simulation workloads, and SASH (Section 5.5) adds speculation to DASH to achieve selective execution. ASH’s memory system also differs from conventional multicores: it does not use global coherence; it performs all inter-tile communication through task arguments; and it performs task-driven instruction prefetching (Section 5.6) to cope with the high instruction footprints of RTL simulation.

Our ASH compiler implementation is based on Verilator, the state-of-the-art Verilog/SystemVerilog simulator [215]. Verilator supports parallelization on shared-memory multicores, which as we have seen, achieves limited speedups (Section 5.2.2). We thus contribute new compiler techniques that automatically divide work into fine-grain dataflow tasks, partition these tasks and their data across tiles, and modify these tasks for selective, speculative execution. We explain these compiler contributions as we introduce DASH and SASH.

5.4 Prioritized Dataflow Execution with DASH

This section presents the DASH system, which implements dataflow execution but not selective execution—all tasks run in every simulated

cycle. We first present DASH’s execution model and ISA; then describe its hardware implementation; and finally explain how RTL compiles to DASH.

5.4.1 Execution model and ISA

As in Swarm, all computation in DASH happens through *tasks*, which execute a unit of code—a C++ function in our implementation. Tasks can have one or more input *arguments* which act as the tokens in dataflow execution (Section 2.1). Tasks produce arguments which are consumed by other tasks.

Dataflow execution: Hardware provides native support for dataflow execution. A task sends arguments to other tasks with a `push_args` instruction. Hardware buffers the arguments for each task; the task becomes *ready* when all its arguments are available. Ready tasks are queued and dispatched to cores for execution. Cores execute each task to completion, and stall if no further tasks are available.

Hardware support for dataflow execution avoids communication through shared memory: a task enqueues output arguments from registers, and when a task starts execution, its input arguments are placed in registers.

Prioritized execution and timestamps: *Prioritized execution* is DASH’s key distinguishing feature. Previous dataflow architectures (Section 5.10.2) execute the dataflow graph in an unordered fashion. This often produces an excessive number of in-flight task arguments, requiring large storage structures for arguments and associative lookups [14, 256].

Instead, DASH prioritizes tasks for execution: software assigns a unique integer *timestamp* to each task, and hardware task units dispatch ready tasks for execution in relaxed timestamp order. The DASH compiler assigns each task a unique timestamp to identify distinct tasks as well as to prioritize work in the critical path and keep a small memory footprint. Thus, DASH has much lower memory footprint than prior dataflow architectures, and a simpler implementation (Section 5.9.3).

Timestamps also differentiate invocations of the same task. Each task invocation has a different timestamp (as each task runs once per simulated cycle), and timestamps enable their arguments to coexist. This allows overlapping the execution of successive cycles, increasing parallelism.

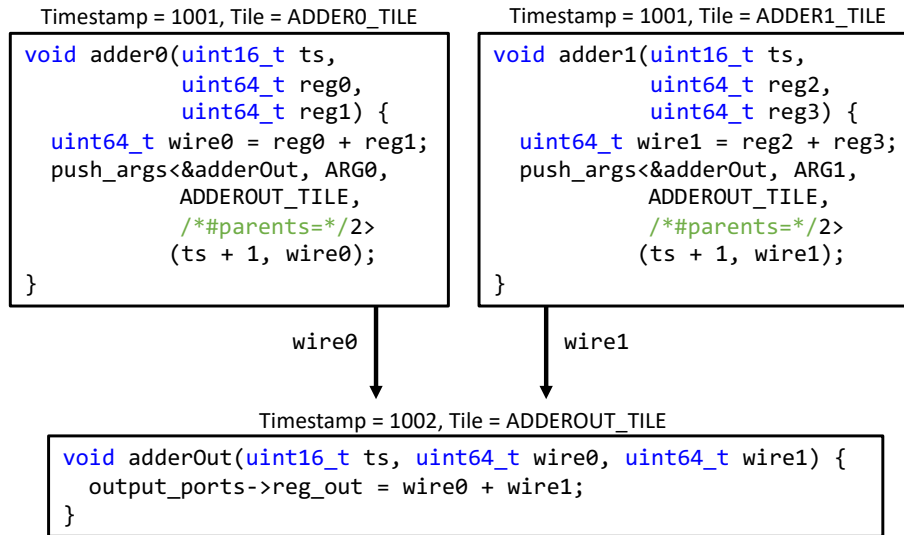


Figure 5-5: Task code for the adder tree in Figure 5-1.

Memory: Although task arguments are passed through registers, tasks can also access memory. Supporting data accesses serves two purposes. First, some simulated structures, like SRAM memories, have a lot of state. It would be inefficient to encode all this state in dataflow edges; it is better to store this state in memory and have dataflow nodes that access it. Second, hardware limits the number of arguments per task (our implementation allows up to five 64-bit arguments). Tasks that exceed this limit must store some inputs in memory.

Task mapping: While memory stores some state, there is no global shared memory, and *all* global communication happens through task arguments. To enforce this restriction, each task is mapped to run on a fixed tile, and all memory accesses to the same data must happen from same-tile tasks.

Argument enqueue ISA: The `push_args` instruction conveys the recipient task’s metadata: function pointer, timestamp, tile id, and the total number of arguments. `push_args` may also convey one or more argument values.

Figure 5-5 shows this interface, with code for Figure 5-1’s adder tree. Tasks `adder0` and `adder1` can run in parallel, each enqueueing arguments for the `adderOut` task to sum.

5.4.2 DASH hardware implementation

The organization of DASH’s task unit is shown in Figure 5-6. It consists of an Argument Queue (AQ) and an Argument Send Buffer (ASB), which resemble Swarm’s Task Queue (TQ) and Task Send Buffer (TSB). But these structures now handle arguments that must be combined in order to execute a recipient task. DASH has no commit queue as it executes all tasks non-speculatively, and we will discuss how SASH handles speculation later.

Each execution of `push_args` creates an *argument descriptor*, composed task arguments and metadata for the recipient task (function pointer, tile id, timestamp, and number of parents). This descriptor is sent to the tile where the task is mapped, and queued in the AQ. The task unit merges queued descriptors of ready tasks, and dispatches them for execution.

Much like Swarm’s TSB, ASBs buffer newly produced data and autonomously sends it to other tiles. This makes `push_args` asynchronous: the core continues execution and does not stall while the descriptor is sent. ASBs use exponential backoff to handle full AQs.

Each tile’s AQ stores in-flight argument descriptors for tasks mapped to the tile. Along with the argument descriptors, the AQ also has a priority queue sorted by timestamp. This enables quickly finding the argument descriptors for the lowest-timestamp queued task. We implement this fixed-capacity priority queue using a pipelined heap [2, 27] that can pop one descriptor per cycle, similar to Swarm’s TQ (Section 2.3).

The task unit’s *merge unit* consumes argument descriptors from the AQ in priority order, and merges them into tasks. The merge unit is similar to a reservation station [240] in an OOO core: it has a small number of entries (16 in our implementation). Each entry gathers the arguments for a task: the first descriptor for a task allocates a new entry, and later arguments for the task are merged in the same entry. After all the arguments are merged into one entry, the entry is freed and the resulting ready task is queued for execution at a core, in a small *ready task buffer*, as shown in Figure 5-6. Ready tasks may execute out of order, while tasks with missing arguments wait in the merge unit. Unlike reservation stations, since tasks may arrive and allocate merge unit entries out of order, the merge unit may need to evict an unready task back to the AQ to allow an earlier-timestamp task to be merged.

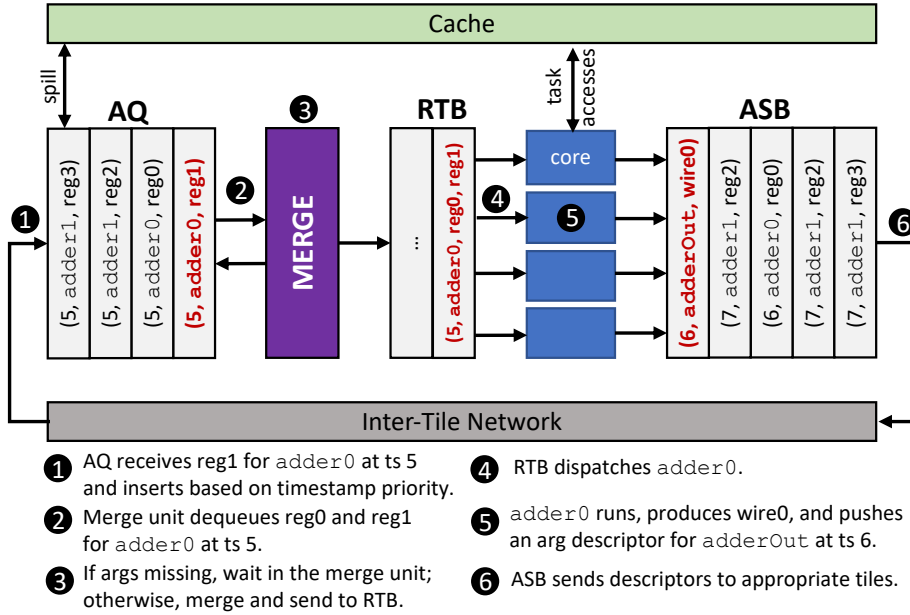


Figure 5-6: DASH tile, with descriptors from Figure 5-5.

These evictions are very rare (Section 5.9.3).

The AQ has a small and fixed size (512 descriptors in our implementation). When the AQ fills up, a simple FSM spills high-timestamp descriptors to memory and later returns these descriptors to the AQ in timestamp order. This prevents high-timestamp descriptors from starving low-timestamp ones. The spiller maintains priority queues in untracked memory similar to Swarm’s generic software spiller (Section 2.3.5), except that requeues are prioritized with timestamp one less than the timestamp of the spilled task. This ensures the low-timestamp argument descriptors are returned to the AQ right before they are needed. Because each task has a unique timestamp and we guarantee each task has a limited number of descriptors (up to 8 in our implementation), forward progress is guaranteed: the argument descriptors for the lowest-timestamp unmerged task will make their way together to the head of the AQ where they can be merged.

Prioritized execution keeps DASH’s merge unit small and efficient. Prior dataflow architectures implemented large *wait-match* units that stored arguments for unready tasks and performed expensive lookups to merge them and find ready tasks [64, 163, 230]. Instead, DASH uses tiny merge units similar to reservation stations, which work almost as well as unbounded ones because most lowest-timestamp tasks have all

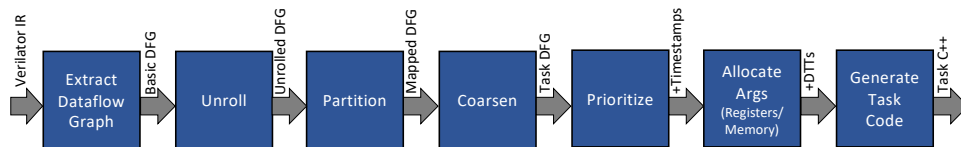


Figure 5-7: DASH compiler passes.

arguments available (Section 5.9.3). Moreover, prior dataflow architectures hinder memory footprint: they execute the dataflow graph in an unordered fashion, so produced arguments may not be consumed for a long time. Instead, prioritization produces arguments in the right order, so modestly sized AQs suffice to keep spills rare.

5.4.3 DASH compiler

We build DASH’s compiler by modifying Verilator [215], a state-of-the-art compiler that transforms Verilog RTL into an efficient parallel C++ simulator. We reuse its frontend, which translates Verilog to a dataflow-style IR. We then heavily modify its backend to produce DASH code from this IR. Figure 5-7 shows our backend’s compiler passes, which we describe next.

5.4.3.1 Increasing parallelism with the unrolled dataflow graph

Existing RTL simulators parallelize execution cycle by cycle, and use a *single-cycle* dataflow graph like the one introduced in Figure 5-1(b). In this representation, wires are converted to dataflow edges, but registers, inputs, and outputs are *in memory*. This representation limits parallelism in two ways. First, it induces write-after-read dependences on registers: every read to the register must be performed before the register is updated. This creates dependences within a cycle. For example, in Figure 5-1(b), the adders in the second pipeline stage must read their registers before the multipliers in the first stage write them. Second, the single-cycle dataflow graph makes it hard to parallelize across cycle boundaries because it assumes that registers are up-to-date at the start of each simulated cycle.

To overcome these limitations, we contribute the *unrolled dataflow graph*. This representation *turns registers into cross-cycle dataflow edges*. Figure 5-8 shows this representation, distinguishing between **same-cycle edges** (wires) and **cross-cycle edges** (registers). This representation

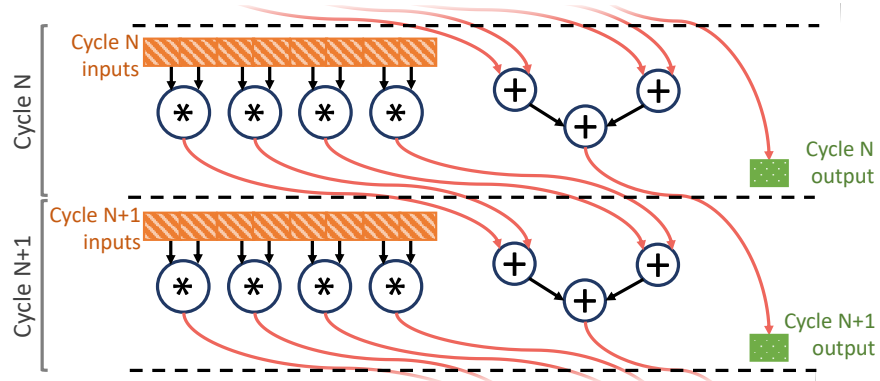


Figure 5-8: The unrolled dataflow graph turns registers into cross-cycle edges (shown in red) to expose more parallelism.

avoids write-after-read dependences and overlaps computations from different simulated cycles, exposing more parallelism.

5.4.3.2 Mapping dataflow nodes

DASH first *partitions* the unrolled dataflow graph across tiles. Since sending arguments across the chip consumes bandwidth and energy, we try to both minimize cross-tile communication and balance work across partitions. In addition, nodes that access the same data in memory (e.g., an SRAM array, as explained in Section 5.4.1) are restricted to the same tile. Finally, we map nodes that incur (rare) system tasks, like I/O, to a fixed tile, which lets us use standard libraries despite the lack of coherence.

We use METIS [116] to perform this partitioning, minimizing the sum of cross-partition edge costs while keeping the sum of node costs per partitions roughly balanced. We estimate a node’s cost as the number of instructions within it and an edge’s cost as the bits of its argument descriptor.

5.4.3.3 Coarsening dataflow nodes into tasks

The compiler then *coarsens* the partitioned graph, producing larger tasks from small dataflow nodes. We modify Verilator’s merging pass (described in Section 5.2.2) to prevent merging dataflow nodes or tasks from different partitions. The merging pass prioritizes merging tasks that incur the minimum degradation in the critical path. Additionally,

we limit the size of resulting tasks using a static cost model, producing tasks with a few tens of instructions on average, enough to amortize DASH’s small task overheads while leaving enough parallelism to keep hundreds of cores busy.

After these passes, we have a *task dataflow graph* where each task is mapped to a tile, and memory accesses are local to a tile: *all cross-tile communication is through descriptors*.

5.4.3.4 Prioritizing tasks

DASH next assigns a unique timestamp to each task, prioritizing their execution within and across cycles. Within a cycle, each task has a *level* l : the longest chain of tasks that produces an argument for the task, measured from the start of the cycle. For instance, assuming that each node in Figure 5-8 is a task, the multipliers and initial adders have $l = 0$, the final adder has $l = 1$. We sort tasks by level and then assign each static task a unique timestamp d that is compatible with level order. These timestamps reflect a topological order in the single-cycle dataflow graph, similar to the timestamps T4 generates (Section 3.5.3). Each dynamic task may given $timestamp = D \cdot cycle + d$, where $cycle$ is the current simulated cycle and D is the number of tasks in the single-cycle dataflow graph. On each `push_args`, the producer task computes the consumer’s timestamp starting from its own. If a producer and consumer have timestamps d_p and d_c , the consumer timestamp is $d_c - d_p$ larger than the producer’s if this is a same-cycle dependence, and $D + d_c - d_p$ if it is a cross-cycle dependence.

Prioritizing by breadth-first levels is simple and produces a good schedule because coarsening produces similarly sized tasks (using finer-grain instruction costs instead of levels produces a similar schedule but makes timestamp computation costlier). When path depths are uneven, this schedule allows starting the execution of the next cycle before the current cycle’s paths are computed, but gives priority to critical paths from the current cycle.

5.4.3.5 Passing arguments through registers and memory

DASH hardware allows each task to receive arguments in up to five 64-bit registers. In most cases, all arguments fit in registers; non-fitting values are read from memory.

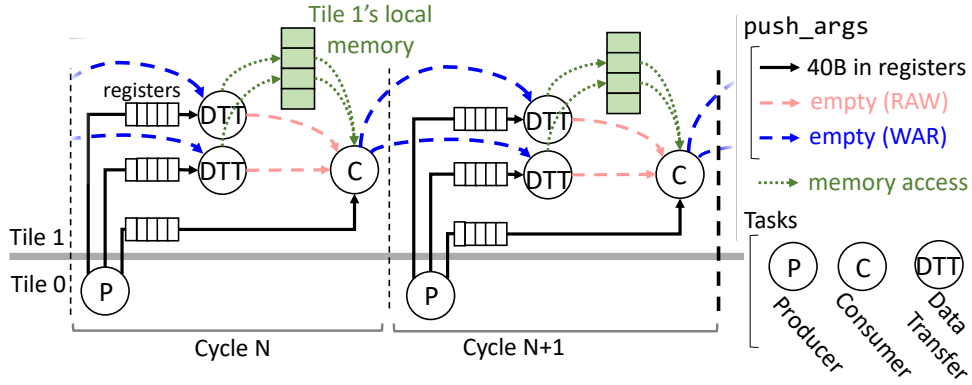


Figure 5-9: Example showing how consumers read non-fitting values from memory, and how DTTs transfer them from producers.

If producer and consumer are on the same tile, the producer simply writes the non-fitting values to memory (at a pre-allocated location specific to the consumer task), and the consumer later reads them. But since ASH does not have global shared memory, when the producer is on a different tile, values are communicated through auxiliary *data-transfer tasks* (DTTs). Figure 5-9 shows DTTs in action: a producer must pass a large 120-byte value. It passes 40 bytes directly to the consumer through arguments, and creates two DTTs to ship the remaining 80 bytes. Each DTT takes 40 bytes of arguments from the producer, and is sent to the consumer's tile, where it writes these bytes to a fixed memory location. Each DTT has a **dataflow edge** to the consumer task (with no arguments) to preserve RAW dependences.

If a producer task must pass an argument to many consumers, the compiler emits a tree of spawner tasks, similar to that in T4's bounded tree expansion (Section 3.6.2), which fans out efficiently across the system to pass these values in parallel to their consumers. This *fan-out* tree is an efficient compiler-generated implementation of multicast messaging that reduces inter-tile communication. ASH hardware also limits the number of direct parents (producer tasks) of each task (to 8 in our implementation). To address this, the compiler emits a *fan-in* task tree for tasks with many arguments. For instance, if in the previous example the producer had to pass a much larger value that required 32 DTTs, these would form a 2-level tree: the consumer would have 8 direct parents, and each of these would have 4 DTTs as parents.

Storing some arguments in a single memory location introduces

write-after-read (WAR) dependences (like we saw in Section 5.4.3.1 for single-cycle dataflow graphs). For instance, in Figure 5-9, the *next-cycle*'s DTTs have a **WAR dependence** with the consumer. For simplicity, we add cross-cycle edges to the task graph to respect WAR dependences, using a `push_args` without arguments from the reader to the writer. We do this *selectively*, adding a WAR edge only if the dependence isn't already covered by existing edges (often, a chain of RAW edges already forces the writer to run after the reader).

The resulting task dataflow graph obeys hardware limits while preserving two invariants: it encodes all dependences as dataflow edges, and localizes all memory accesses.

Finally, the compiler generates C++ code for this task graph, producing a DASH program.

5.5 Selective Dataflow Execution

DASH runs all tasks in the dataflow graph, but many tasks are ineffectual because only a fraction of the logic switches each cycle. SASH leverages low activity factors by running *only tasks whose inputs have changed* on each cycle.

SASH extends DASH hardware to skip inactive tasks and to run tasks *speculatively*, using speculative parallelization as described in Section 5.2.3. We first describe the changes needed for selective execution, then those needed for speculation.

5.5.1 Selective execution

In SASH, parents only push arguments that are different from previous cycles. This requires small hardware changes. First, parents must be able to detect a change in their produced output compared to previous cycles. Second, since each task may only receive *some* of its arguments, it must fill in the missing arguments with their last values. SASH implements these features with simple hardware and two in-memory structures: an *input argument buffer* and an *output argument buffer*.

Ineffectual arguments are filtered: SASH statically allocates a per-task, hardware-managed, in-memory *output argument buffer* that holds the latest arguments the task has pushed. This buffer has finite size because we restrict each task to push at most 8 descriptors, and each

push has at most five 64-bit registers. Each `push_args` reads the output buffer and compares the new and previous arguments. If they differ, the output buffer is updated and an argument descriptor is pushed to the ASB. If they match, no action is taken.

Task dispatch does not wait for all arguments: In SASH, every task that has received at least one argument descriptor is *ready*. The task unit’s merge unit now does *not* store multiple tasks: it merges the descriptors for the lowest-timestamp task and dispatches it for execution.

Dispatched tasks fill missing arguments with old values: Since tasks with inactive parents have missing arguments, SASH maintains a per-task, in-memory *input argument buffer* that records the latest arguments of the task. When a task is dispatched, before starting execution, the input argument buffer is read to retrieve the task’s missing arguments, and then updated with the latest values.

Handling in-memory arguments: When a task consumes inputs from memory (Section 5.4.3.5), two types of descriptors carry no arguments: those that encode read-after-write (RAW) dependences (e.g., between DTTs and consumer in Figure 5-9) and those that encode WAR dependences (e.g., between consumer and next-cycle DTTs in Figure 5-9). When a descriptor has no arguments, `push_args` takes an `isRAW` flag to distinguish them. Argumentless RAW descriptors are not filtered, as they encode a memory dependence. (in this case, software checks whether the in-memory value is the same, and skips pushing the descriptor if so). WAR descriptors are used in speculative execution, described next.

5.5.2 Speculative execution

SASH dispatches tasks even if some of their arguments are missing. Given enough parallelism and appropriate timestamps, most missing arguments arise because their producer task was skipped or filtered them. But occasionally, an argument is missing because it is *late*, so using its old value is incorrect. Therefore, SASH runs tasks speculatively: on a late-arriving argument, the task and all its consumers are aborted, and the task is re-executed with the new argument.

SASH implements speculative execution by adapting techniques from from Swarm and Time Warp [104]. Note that, unlike Swarm and most other architectures for speculative parallelization, SASH does *not* detect

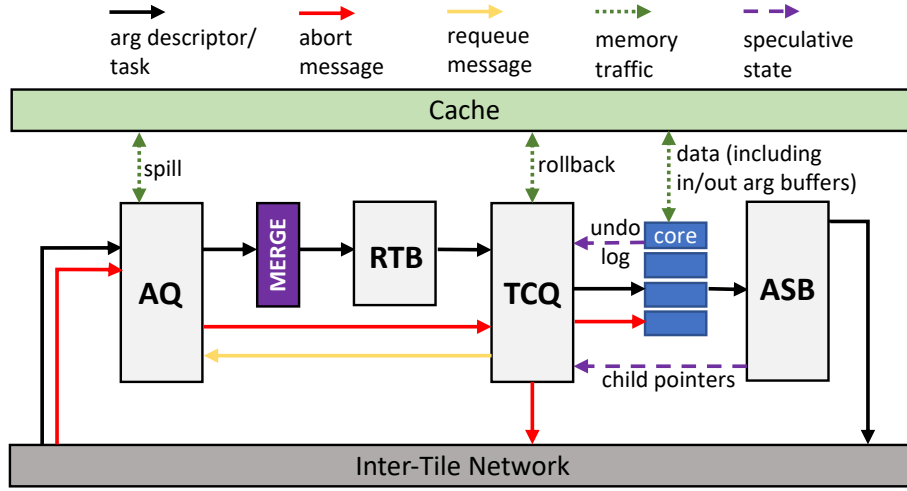


Figure 5-10: SASH tile.

misspeculation through memory accesses. This results in much simpler hardware.

Speculation and rollback: Like Swarm, SASH uses a per-tile *commit queue* (CQ) as shown in Figure 5-10 that holds the speculative state of uncommitted tasks. Each dispatched task is allocated an entry in the CQ. But unlike Swarm’s CQs, SASH does not need to store address sets in Bloom filters to facilitate conflict detection.

SASH detects conflicts when descriptors arrive at the tile. The CQ is checked to find if a task with the same timestamp was dispatched. If so, the task is aborted and requeued to the AQ, along with the new argument.

WAR descriptors are handled differently: they undergo conflict detection as described above, and then they are immediately discarded, without ever being enqueued in the AQ. This is because WAR descriptors do not carry any data for a true data dependence, they ensure that writers do not clobber earlier readers. By conflict-checking but discarding the descriptor, we enforce its WAR dependence but prevent an ineffectual task from running.

SASH adopts eager versioning with undo logs to handle memory accesses, following Swarm’s implementation: on a store, the core updates memory in-place, and pushes the old value to a small in-memory undo log. If the task aborts, memory values are restored from the undo log. The undo log is discarded when the task commits. Note that updates

to output and input argument buffers are also logged.

Finally, SASH implements cancellation of pushed argument descriptors as in Time Warp. When a task aborts, in parallel with restoring memory, its CQ deletes each argument descriptor that the aborted task had pushed, sending messages to perform this deletion if the argument descriptor is in a remote AQ. If the argument descriptor was used by a dispatched task, that task instance is aborted. The AQ overflow mechanism from DASH is modified to use the spill policy from Swarm, spilling only argument descriptors whose parents have committed.

High-throughput commits: To make commits fast, each CQ can buffer tens of finished tasks per core, and the task unit determines which tasks are safe to commit in bulk, amortizing commit overheads over many tasks. Specifically, we adopt the virtual-time algorithm from Time Warp, with some changes from how it is used in Swarm that exploit the uniqueness of timestamps. As in Swarm, periodically (every 100 ns in our implementation) each tile sends the timestamp of its earliest unfinished task to a global arbiter, which finds the minimum such value across all tiles and broadcasts it. CQs then commit all tasks with timestamps lower or equal than this minimum. Unlike in Swarm, unmerged argument descriptors in ASBs are treated as unfinished tasks with timestamp one lower than their destination task. This prevents misspeculating task executions from committing when an argument descriptor that needs to be merged to that task has yet to arrive at the destination AQ to be merged. This does not violate the rule that the virtual time broadcast by the commit arbiter must monotonically increase, because at the moment any argument descriptor enters the ASB, its producer must be executing, and that producer must have a lower timestamp because the ASH compiler assigns each task a unique timestamp.

5.6 Task-Driven Instruction Prefetching

Beyond dataflow and selective execution, ASH includes a simple optimization to cope with high instruction footprints: large hardware designs execute multiple megabytes of code per simulated cycle. On conventional systems, this causes frequent instruction cache misses that limit performance [17].

To avoid this problem, we add a simple *task-driven instruction prefetching* mechanism that fetches the task’s code into the core’s

instruction cache ahead of execution. We add a small Ready Task Buffer (RTB) to each core. The task unit pushes tasks to this RTB eagerly, so the RTB has the task that runs *after* the currently executing task. The core starts fetching a task’s code into the instruction cache as soon as the RTB receives it. To do this accurately, we include the task function’s size (up to 8 lines) in some of the unused high-order bits of the task pointer.

Task-driven instruction prefetching virtually eliminates instruction cache stalls and avoids overfetching, because tasks have a few tens of instructions each and execute most of them (branches do not usually skip whole cache lines).

5.7 Hardware Costs

We evaluate an ASH implementation with 256 scalar in-order cores—for RTL simulation, using many simple cores that are closely integrated is preferable to using fewer complex cores, as we will see later. We use a 2-level memory hierarchy with 1 MB tile-private L2s and quad-channel DDR5 memory, shown in Figure 5-4 and with parameters given in Table 5-3.

Component	Area (mm ²)
256 cores	45.1
64×1MB L2s	39.3
4×Mem ctrl+PHY	25.0
64×SASH task units	5.6
Total	115.0

Table 5-2: Area breakdown.

Table 5-2 shows the estimated area of the SASH system in a 7nm process, which takes a modest 115mm². We estimate core area using scaled-down Atom Bonnell cores [262], and other components are measured from Alder Lake-S [199] (built in Intel 7). We estimate SASH’s area requirements (DASH would be simpler) by synthesizing its key components based on the Chronos open-source RTL [238] using yosys [253] on FreePDK45 [160] and scaling to 7nm. SASH adds 45 KB of state per tile and adds 4.8% to chip area, a modest overhead.

5.8 Methodology

We evaluate DASH and SASH by extending the Swarm simulator. In particular, we model ASH’s task units, and we modify the cache and on-chip network models to simulate a partitioned memory hierarchy, and

Cores	256 scalar in-order cores in 64 tiles (4 cores/tile), 2.5 GHz, scoreboarded, 16-entry ld/st queue, x86-64 ISA
L1 caches	16 KB L1D & 16 KB L1I per core, 8-way, 2-cycle latency
L2 caches	1 MB per tile, 16-way, inclusive, 9-cycle latency
NoC	16×16 mesh, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [251])
DRAM	4 controllers at chip edges, 120-cycle minimum latency
ASB	holds 64 argument descriptors per tile
AQ	holds 512 argument descriptors per tile
Merge	16-entries per tile to merge descriptors, DASH only
RTB	1 entry per core holding merged task for dispatch
CQ	512 entries per tile for SASH to track speculating tasks, global Virtual Time computed every 100 ns

Table 5-3: Parameters of the evaluated systems.

we modify the commit protocol to support SASH. We reuse Swarm’s detailed cycle-level models for cores and main memory.

To limit excess speculation, we implement a gentle hardware heuristic that prevents SASH tasks from running far too early, when they are almost certain to abort. In particular, software registers to hardware the range of distinct timestamps D used by tasks within one simulated cycle (Section 5.4.3.4). Then hardware prevents any SASH task from being dispatched from the RTB to a core if the difference between the task’s timestamp and the timestamp of the most recently committed task is greater than D . This prevents tasks running more than one full simulated cycle ahead. This prevents aborts that could happen due to either a new argument descriptor arriving later for the same task, or a previous argument descriptor that was merged into that task being canceled because its parent aborted.

Baselines: We compare with Verilator running on two systems: a 32-core AMD Zen2 CPU (Threadripper 3975WX) at 3.5 GHz (also used in Section 5.2); and a simulated multicore with the same parameters in Table 5-3, but no ASH features and a shared, coherent LLC (instead of tile-private L2s).

Benchmark hardware designs: We simulate the four large, complex, and diverse hardware designs in Table 5-4:

(1) *Vortex GPU system:* We use the open-source Vortex GPU [239] with 32 cores using 2 lanes and 4 warps each, running an OpenCL kernel that performs vector addition.

	Vortex	Chronos/PE	Chronos/RV	NTT
Description	Full-system	Graph	RISC-V	Cryptographic
	GPU	accelerator	manycore	accelerator
Nodes	245.5K	297.1K	375.9K	29.4K
Edges	414.7K	635.0K	733.2K	41.7K
Tasks	86.3K	82.7K	65.5K	7.4K
% DTTs	14.9 %	57.4 %	51.3 %	0.0 %
Task Edges	167.0K	207.0K	184.0K	11.8K
Parallelism	672	407	415	558
Activity Factor	7.1 %	17.4 %	15.0 %	97.0 %
1-core Cycles	8.6M	8.4M	8.9M	0.8M
Code Footprint	10.1 MB	16.2 MB	13.7 MB	0.7 MB
Compile Time	70.6s	140.5s	134.1s	7.2s

Table 5-4: Main characteristics of the benchmark hardware designs.

(2) *Chronos domain-specific accelerator*: We use the Chronos open-source framework [238] to generate an accelerator for Dijkstra’s algorithm for shortest weighted paths. We simulate a 128-PE system traversing a 128×128 grid.

(3) *Chronos RISC-V manycore*: To represent manycore systems, we use another Chronos configuration to generate 128 simple VexRiscv [221] cores instead of specialized PEs.

(4) *NTT wide functional unit*: We evaluate a pipeline that computes Number Theoretic Transforms (NTT), an FFT variant using modular arithmetic and a key component of Fully Homomorphic Encryption accelerators [123, 202, 203]. We follow CraterLake’s NTT unit design [203], which we implement from scratch. This unit performs 256-point NTTs using a 256-wide, 8-stage pipeline with 1024 modular multipliers.

These designs have diverse activity factors (Table 5-4) and are large, requiring multi-FPGA emulation (from 2 FPGAs for Chronos/PE, as we saw in Section 5.2.4, to 6 FPGAs for NTT).

RTL compiler implementation: Our implementation adds 12 K lines of code (LoC) to Verilator, and the whole compiler takes 86 K LoC of C++. We retain Verilator’s fast compilation times: each design takes 7.2–140s to compile (Table 5-4).

Verilator is a full-featured Verilog/SystemVerilog simulator, and benchmarks show that it matches or outperforms state-of-the-art commercial simulators, like VCS [61, 216, 217]. Verilator has fewer features than commercial simulators, e.g., no support for 4-state logic, VHDL,

or private IPs. However, these aren’t ASH limitations: ASH provides general-purpose cores, and could accelerate other simulators.

Sampling: Each RTL simulation runs for many billions (often trillions) of instructions, so we use a sampling-based approach following Sim-Points [210]. First, we functionally execute the whole RTL simulation to record activity factors. Then, we select three cycle ranges representative of the full program. We fast-forward to the start of each range, warm up for 3 simulated cycles, and gather results for the remainder of the range. The methodology ensures that our reported results have the same activity factor as the full run. Even with sampling, each simulation runs many millions of instructions.

5.9 Evaluation

System	Design	Vortex	Chronos/ PE	Chronos/ RV	NTT	gmean
Zen2 t=1		3.8	3.1	2.8	101.0	7.6
Zen2 Best		t=24 14.8	t=16 10.7	t=12 9.2	t=1 101.0	19.6
Baseline t=1		0.2	0.3	0.2	2.6	0.4
Baseline Best		t=64 25.6	t=64 19.3	t=64 18.8	t=16 86.0	29.9
DASH 256-core		201.3	108.1	133.0	1,644.7	262.7
SASH 256-core		686.5	413.9	340.5	1,689.2	635.8
SASH/Zen2 Best		46.5×	38.6×	36.8×	16.7×	32.4×
SASH/Baseline Best		26.8×	21.5×	18.1×	19.6×	21.3×

Table 5-5: Workload simulation speeds (in KHz) and speedups.

5.9.1 ASH widely outperforms baseline systems

Table 5-5 reports simulation speeds. Each row reports results for a system, and each column shows the speed for a specific benchmark design. Speeds are in KHz, i.e., thousands of simulated cycles per second. The last rows report SASH’s speedups over the baselines. Overall, SASH is 32× faster than Zen2 despite taking 3× less area than Zen2, and it is 21× faster than the simulated 256-core baseline.

Parallel Verilator has limited scalability, so for the baselines, Table 5-5 reports both their 1-core and best parallel speeds, showing the best thread count on each cell (e.g., Zen2’s best Vortex result is with 12

threads). DASH and SASH always improve with more cores, so Table 5-5 reports only their 256-core speeds. The Zen2 system achieves a gmean speed of 19.6 KHz, whereas the simulated baseline multicore achieves 29.9 KHz. Note that, for the simulated results, smaller systems have fewer tiles and thus less LLC. Thus, at 1-core, Zen2 widely outperforms the simulated baseline mainly due to having much more cache; but on the best results, which have a similar amount of LLC, the simulated baseline outperforms Zen2. This happens because Verilator’s makes limited use of out-of-order cores: its large code footprint and poor branch predictor performance limit IPC below 1.0, so using simpler cores is preferable.

DASH and SASH achieve high simulation speeds: 263 KHz and 636 KHz gmean, respectively. DASH widely outperforms the baselines because it (1) exploits more parallelism; (2) avoids synchronization overheads due to hardware support; and (3) reduces memory stalls due to tile-private caches, distributed tasks, and prefetching. SASH further outperforms DASH due to selective execution. SASH’s improvement is highly correlated with the design’s activity factor (Table 5-4).

Looking across benchmarks, Table 5-5 shows that Vortex and the two Chronos variants achieve similar simulation speeds, e.g., around 10 KHz on Zen2. SASH has more pronounced differences across designs due to varying activity factors (Vortex is only 7%, whereas Chronos/RV is 15%). The NTT benchmark is quite different from the others: despite being the largest design in terms of area, it is faster to simulate and has a lower code footprint (Table 5-4), as it is dominated by arithmetic operations. The NTT benchmark also has a near-100% activity factor, representing a worst-case scenario for SASH. The fact that SASH and DASH perform similarly on this benchmark shows that SASH performs selective execution efficiently.

5.9.2 Architectural analysis

We now present more detailed data to understand the performance and efficiency of DASH and SASH, and evaluate the impact of our contributions.

Scalability: Figure 5-11 shows the speedup of all simulated systems as we scale from 1 to 256 cores. Larger systems have more tiles and thus more total cache capacity (scaling this way keeps the area ratio between cores and cache fixed).

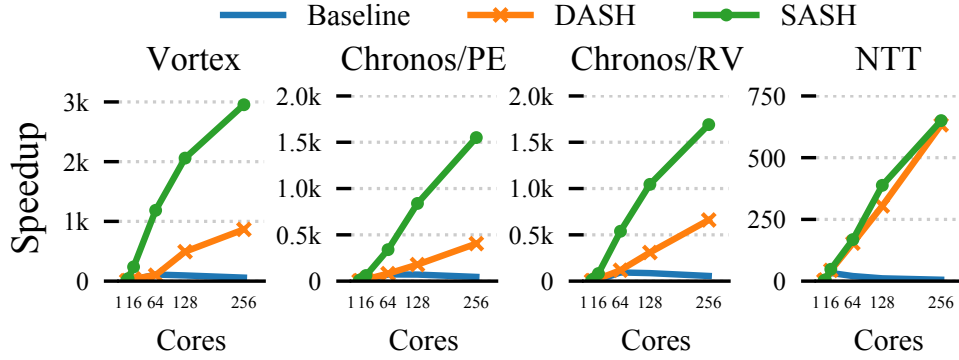


Figure 5-11: Speedups of DASH and SASH over serial Verilator on the 1-core simulated baseline as cores grow from 1 to 256.

Compared to serial Verilator running on one core, 256-core DASH is gmean $617\times$ faster, and SASH is gmean $1,485\times$ faster (up to $2,877\times$ on Vortex).

Figure 5-11 shows that speedup increases linearly throughout the range for DASH, showing that it leverages the plentiful parallelism in these benchmarks. SASH also shows good scalability, though the slope of the speedup curve drops somewhat between 128 and 256 cores for applications with a low activity factor (Vortex and the Chronos variants). SASH performs less work per cycle, which tends to reduce dynamic parallelism (the ratio of active work to critical path).

Cycle breakdowns: Figure 5-12 gives more insight into these results by showing how cores spend cycles for SASH systems of different sizes. Each plot reports results for a different benchmark, and each bar reports the number of cycles summed across all cores, normalized to the cycles of the 1-core system (lower bars are better, and a height of 1.0 denotes a speedup equal to the number of cores). Each bar is broken down into the cycles that cores spend (1) running tasks that later **commit**, (2) running tasks that later **abort**, and (3) **idle** because the AQ is empty or the CQ is full.

Figure 5-12 shows two key trends. First, the number of committed cycles *decreases* as the system size grows, with a sharp knee around 16 or 64 cores. This happens because L2 misses and memory stalls fall as the system grows: each tile executes a smaller fraction of the tasks, and the tasks' working set fits into the L2 caches for the larger systems. This causes IPC to grow from about 0.2 at 1 core to about 0.6 at 256 cores. Specifically, these models have both a large code footprint and a

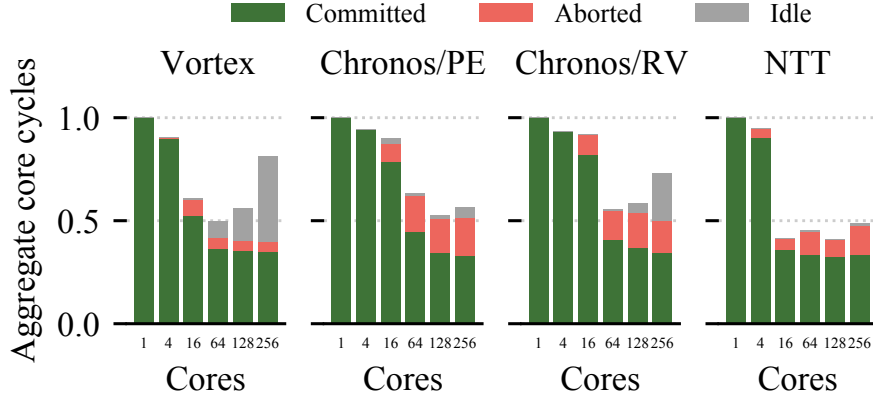


Figure 5-12: Breakdown of core cycles for SASH on 1–256 cores.

substantial data footprint; this speedup arises because the model’s *data* starts fitting in-cache (instruction footprint is even larger, but those are prefetched, as we will see next).

In addition, Figure 5-12 shows that most time is spent executing committed tasks. Aborts are rare, and the slight scalability drop at 256 cores in Vortex and Chronos/RV stems from idle cycles due to lack of work.

5.9.3 Impact of ASH features

Prioritized dataflow: ASH’s prioritized dataflow execution has two benefits over prior dataflow architectures: simplifying hardware, and keeping memory footprint in check (Section 5.4.2). We now analyze these benefits.

Figure 5-13 shows DASH’s gmean performance as merge unit entries grow, relative to the default (16 entries). While small merge units incur some degradation (e.g., 8 entries is 3% slower), the default is sufficient, and a merge unit with unbounded entries is only 3% faster. Thus, prioritized tasks enable dataflow execution with minimal resources (each tile’s merge unit is much smaller than the issue queue of modern OOO cores).

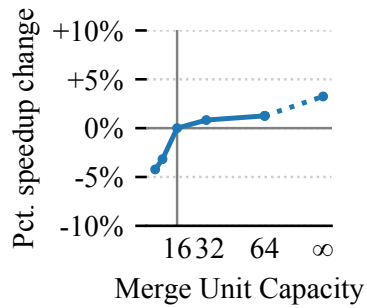


Figure 5-13: Sensitivity to DASH merge unit size.

Unlike in OOO cores, arguments arrive at a tile out of order, and a late-arriving task might evict a higher-

timestamp task from the merge unit. Such evictions are rare: one per 500 dequeues on average (thousands of cycles between evictions).

Figure 5-14 compares the memory footprint of prioritized vs. unordered (conventional) dataflow. Unordered dataflow execution runs each task as early as its inputs become available. This causes tasks within short paths in the simulated design (the common case) to run ahead of critical paths, producing arguments that won't be consumed for a long time. Figure 5-14 shows that this is a major problem: conventional dataflow increases average data footprint by $16.8\times$ gmean and by up to $47\times$ (Chronos/PE) over prioritized dataflow. With data footprint in the tens of megabytes (Table 5-4), this blowup would cause substantial main memory traffic, hindering performance.

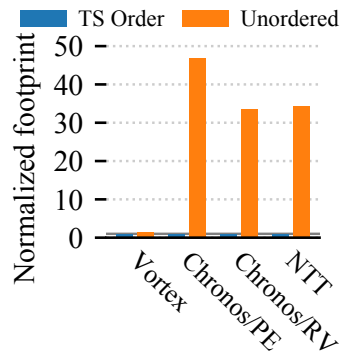


Figure 5-14: Prioritization reduces memory footprint.

Instruction prefetching: So far, all the results we have presented include task-driven instruction prefetching. Figure 5-15 shows the speedup that prefetching achieves on SASH. (DASH results are similar.) Each bar shows gmean speedup across benchmarks for a specific core count. Smaller systems have greater need for prefetching because less code fits on chip. At 1 core, most code is evicted every simulated cycle. At 256 cores, Vortex, Chronos/PE, and Chronos/RV still have 3-6% of L1 instruction cache misses also miss in the L2, chiefly due to L2 cache conflicts. Thus, prefetching code offers speedups across system sizes (e.g., $1.9\times$ at 256 cores). We observe a peak memory bandwidth of 114GB/s at 256 cores (on Chronos/PE), with code making up the vast majority of memory traffic.

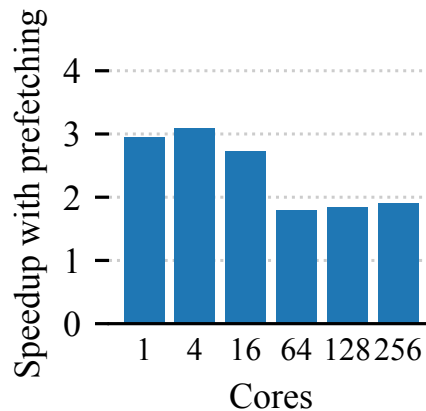


Figure 5-15: Speedup of task-driven instruction prefetching for SASH on 1–256 cores.

Queue utilizations: Figure 5-16 reports the average occupancies of the per-tile Argument Queue (AQ) and Task Commit Queue (CQ) on the 256-core SASH system. Each group of bars reports results for a single benchmark. Each structure holds up to 512 entries. AQ occupancy corresponds to the average number of in-flight arguments per tile, and CQ occupancy is the average number of uncommitted tasks per tile.

The modest AQ occupancy shows that prioritized dataflow execution works well: thanks to ordering tasks by timestamp, we execute the dataflow graph in an order that incurs a modest number of in-flight arguments that comfortably fit in the AQs (three of the designs never spill argument descriptors to memory at 256 cores, and Vortex has minimal spilling). The CQ occupancy shows that tiles exploit out-of-order execution, running ahead of the earliest unfinished task to extract speculative parallelism.

Factor analysis: Figure 5-17 shows gmean speedup vs. parallel Verilator as we add features, quantifying the benefits of our hardware and compiler techniques. *+hw df* uses hardware dataflow execution within each cycle. *+unroll* uses our unrolled dataflow graph instead, which passes register values through dataflow edges and increases parallelism. *+mapping* uses our task partitioning and coarsening algorithm instead of Verilator’s; this configuration is DASH. Finally, *+selective* adds selective execution; this is SASH. Figure 5-17 shows that each of our contributions yields substantial performance gains.

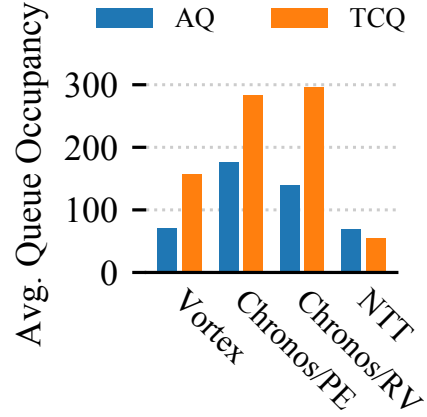


Figure 5-16: Average queue occupancies per benchmark in 256-core SASH.

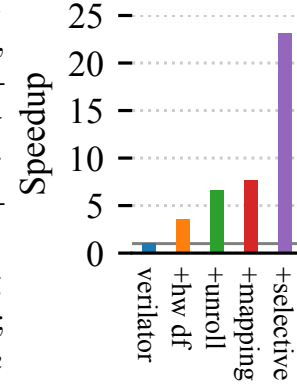


Figure 5-17: Factor analysis.

5.10 Additional Related Work

We first discuss prior architectures with task-level speculative and dataflow execution and compare them with SASH and DASH, then

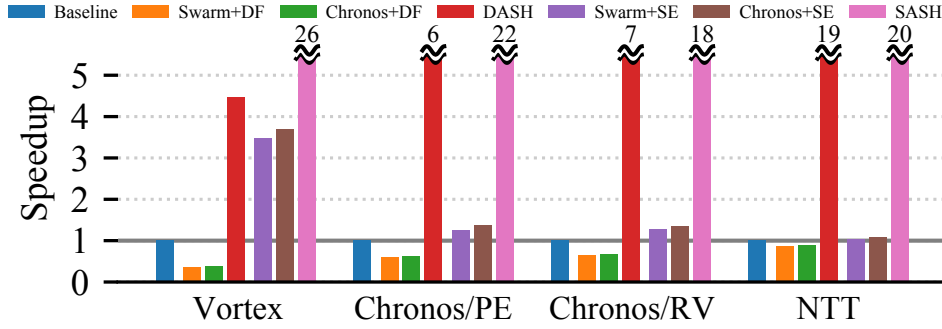


Figure 5-18: Speedup over best Verilator Multi-Core run.

discuss other simulation-related techniques.

5.10.1 Prior task-level speculative architectures

Chronos [2] and Swarm [107] are the closest architectures to SASH. Swarm is a shared-memory multicore with hardware support for small ordered atomic tasks. Tasks can access arbitrary memory, and cache coherence is used to detect conflicts, similar to other speculative systems like HTM [84, 90, 156] and TLS [193, 218, 226]. Chronos avoids cache coherence by mapping each task to a fixed tile and providing hardware support for mutual exclusion. These systems target graph analytics and other domains, including non-RTL event-driven simulation, as they are good match for Time Warp’s [104] optimistic parallelization.

SASH’s speculation support borrows heavily from Chronos (Section 5.5, Section 5.7). However, these systems are poorly suited for RTL simulation because *they do not support dataflow execution*: each task has a single parent and can run the moment it is created. This requires implementing dataflow execution in software, which adds substantial overheads.

To quantify these overheads, we adapt the Swarm simulator [231] to simulate 256-core Swarm and Chronos systems with similar parameters to ASH. Since these systems lack hardware support for dataflow execution, we modify our compiler to do dataflow synchronization in software. Each producer spawns a task for the consumer. All tasks (except the last) write their arguments to memory and finish. The last task gathers all arguments and runs the consumer once. For selective execution, we perform output filtering in software.

Figure 5-18 compares the performance of the baseline multicore, Swarm and Chronos with dataflow non-selective (DF) and selective (SE) execution, DASH, and SASH. Each group of bars shows, for one benchmark, the speedup of each scheme vs. the best-thread baseline. Swarm-DF is *slower* than the baseline, because the lack of dataflow support adds large overheads: instruction footprint grows by about $2\times$, and branches and memory accesses become more frequent, limiting IPC. Swarm-SE has limited improvements over Swarm-DF, and is significantly faster only in Vortex, where the activity factor is lowest. This happens because selective execution in software adds even more overheads. Chronos shows similar trends to Swarm: whereas Swarm uses a coherent cache hierarchy (like the multicore baseline), Chronos uses private caches and avoids coherence (like ASH); this makes Chronos somewhat faster than Swarm. But these systems lack *hardware support for dataflow execution, which is crucial* to accelerate RTL simulation: DASH/SASH are gmean $12.5\times/12.9\times$ faster than Chronos-DF/SE, respectively.

5.10.2 Dataflow architectures

Dataflow execution [55] runs operations as their inputs become available, instead of following a fixed operation sequence. Dataflow execution is a general principle to exploit fine-grain parallelism, and there is rich prior work applying it to a wide range of architectures [55, 64, 163, 165, 166, 176, 230, 256].

Task-superscalar [64] is perhaps the closest to DASH. It performs task-level dataflow execution, with dedicated on-chip structures to track in-flight arguments and dispatch tasks. As we’ve seen (Section 5.4.2), DASH’s key distinguishing feature is *prioritized* dataflow execution: DASH gives each task a timestamp, and follows this priority at runtime. Section 5.9.3 showed that this addresses the key challenges of dataflow architectures. First, prior systems require large on-chip structures to track arguments, e.g., multi-megabyte eDRAMs in Task Superscalar [64], or add metadata to main memory as in I-Structures [15]. They also perform costly associative lookups to find ready operands. Instead, DASH performs dataflow execution on a tiny window of tasks (the number of merge unit entries). Second, unordered dataflow systems often produce many live in-flight arguments, sacrificing locality [14]. Section 5.9.3 showed order-of-magnitude footprint blowups in in-flight

arguments, which ASH avoids.

5.10.3 Hardware emulators

Rizzatti [195, 196, 197] reviews the rich history of hardware emulation, including key technical and commercial developments. IBM’s YSE [179] and EVE [24] were the first specialized architectures for emulation, including ASIC processors to simulate 4-input logic gates and storage elements. Early commercial emulators combined tens to thousands of off-the-shelf FPGAs [18, 120, 126], interconnected with a low-latency network, and software to partition and map large circuits across FPGAs. Current processor- and FPGA-based emulators, described in Section 5.2.4, all follow this template.

ASH fundamentally differs from hardware emulators in many ways: (1) Emulators synthesize RTL to *gates*, which forces the use of specialized processors or FPGAs; instead, ASH compiles RTL to general-purpose code, where each instruction can simulate many gates. (2) Emulators directly map gates to hardware elements so emulator size limits the size of the emulated system; instead, ASH simulates large systems by running tasks over time. Consequently, (3) emulators are large and expensive, with costs in the millions of dollars, whereas ASH systems can be small. (4) Due to these design choices, emulators suffer from long compilation times, as Section 5.2.4 showed. Finally, (5) emulators do not leverage selective execution, and cannot avoid ineffectual work.

5.10.4 Other simulation accelerators

Prior work has also proposed several FPGA-based simulation accelerators, but these systems do not target RTL simulation. RAMP Gold [234] and FAME [235] simulate a multicore system by offloading core simulation to FPGAs, time-division multiplexing cores as needed, and use a high-level timing model for the memory system. RAMP Blue [124] emulates distributed message-passing architectures. These systems are restricted to multicore designs.

Diablo [233] and FireSim [115] emulate datacenter-scale systems and offer scalable multi-FPGA emulation. However, these systems target simulating large collections of loosely connected systems, and use conservative (CMB-style) parallelization. Thus, while these systems work well for scale-out systems (e.g., many simple in-order cores connected

through Ethernet), they cannot simulate a large RTL model that doesn't fit in an FPGA. In addition, adapting existing designs to these systems requires manual changes to the RTL.

5.11 Summary

RTL simulation is a key bottleneck in chip design, and existing simulators and emulators have serious drawbacks. By leveraging classic parallelization approaches (dataflow and speculative execution) and though several novel techniques, we have dramatically accelerated RTL simulation while retaining its fast compilation time and flexibility.

6 Conclusion and Future Work

This dissertation has shown that computer systems can effectively parallelize challenging applications through carefully co-designed compiler and hardware techniques. These techniques bring large performance improvements by enabling applications to use the many cores that manufacturers can already build. We have targeted difficult-to-parallelize applications that have been left behind by current parallel computer systems. By doing so, we have developed parallelization techniques that generalize beyond the limited types of computation that are successfully parallelizable today.

Chapter 3 – T4 targeted general sequential programs, developing new parallelization techniques that are not restricted to particular program regions or control-flow structures. T4 composes fine-grained nested parallelism across arbitrarily complex nested control flow and loop nests, spawning tiny tasks in parallel using task trees. T4 also automatically exploits locality in arbitrary memory access patterns, even those that arise from traversing pointer-based structures. We demonstrated the generality and effectiveness of these parallelization techniques by implementing them in the LLVM compiler and evaluating them with large and hard-to-parallelize C and C++ programs, which we are the first to scale to tens of cores.

Chapter 4 – S5 developed further hardware and compiler techniques that allow a programmer or high-level compiler to annotate challenging data structures so the system can optimize these structures with knowledge of the structure’s purpose in scheduling deferred computation. Like T4, S5 preserves sequential semantics and exploits fine-grained parallelism without restricting the operations users may perform in parallelized tasks. S5 demonstrated that small dedicated hardware queues need not be specialized to an application to exploit specialized software optimizations that deliver order-of-magnitude efficiency improvements

over state-of-the-art parallel systems. As a result, S5 scales several challenging applications linearly to hundreds of cores while maintaining the simplicity of writing sequential code.

Chapter 5 – ASH adapted techniques from Swarm and T4 into an accelerator specialized for RTL simulation. ASH exploits static knowledge of dependences with novel and simpler hardware mechanisms for speculation with support for dataflow execution. ASH demonstrates that T4’s principles of compiling existing workloads into fine-grained tasks with task trees for high-throughput spawns can be made efficient and work with specialized hardware, even hardware that does not support shared memory with cache coherence. ASH used speculation to enable skipping redundant work and communication, delivering order-of-magnitude efficiency improvements over non-speculative dataflow execution. The ASH compiler fully automates the use of these optimizations. As a result, a small ASH accelerator is $32\times$ faster than Verilator running on a complex multicore CPU.

T4, S5, and ASH take several large steps forward on a path where hardware and compilers take responsibility to make parallelization easy and mostly or entirely automated. After evaluating our techniques on challenging workloads, we expect our underlying principles will apply to general computation, paving a path for parallelization to become both practical and ubiquitous.

6.1 Future Work

Several avenues exist for future work to further improve the large speedups demonstrated in this dissertation, and to bring these performance benefits to more workloads in the real world.

Distributed or disaggregated systems: The systems we have presented all tightly integrate their processor cores onto a single chip with a fast on-chip network and directly attached DRAM. However, rapid advancement in interconnect technologies is causing our industry to trend towards building large systems that distribute resources such as memory or accelerators that are distributed over a large network. This enables systems that scale out a single computation to thousands of chips [113]. This opens up the exciting opportunity to explore scaling up systems for speculative parallelization from mere hundreds of cores to systems with many thousands of cores, distributed across many chips.

The techniques we have developed for large chips with hundreds of cores may help make it easier to build a large distributed system. For example, we use point-to-point communication through registers that exploit large hardware- and software-managed queues which can hide communication latencies during task spawns. Additionally, ASH’s specialized domain-specific design exploits statically known dependences to avoid the need for cache coherence and uses simpler conflict detection and resolution protocols, placing fewer requirements on the network than cache-coherent Swarm.

New techniques are likely needed when communication between cores goes from taking tens of cycles within a chip to taking thousands of cycles over an off-chip network. Important avenues to explore include identifying what algorithms, data structures, and hardware support are needed to reduce usage of limited off-chip communication bandwidth. This could involve improvements to how our compilers partition tasks and data across chips, which could take into account the non-uniformity of communication latencies between neighboring tiles or chips and those further away, whereas our current techniques map tasks to tiles in a homogeneous fashion. It might also involve implementation techniques such as batching or compressing the communication that occurs when distributing task descriptors between tiles.

Exploiting more application semantics: S5 currently provides a limited set of abstract data types for scheduling, but a more full-featured system could integrate further primitives to allow programmers to exploit other computation patterns. Future work could combine S5 with prior hardware and software techniques for efficient parallelization of particular patterns such as commutative and associative reductions [265, 266] or dynamically growing dictionaries that support random lookups [267].

There are also applications where programmers can easily identify coarse-grained unordered tasks, such as in transaction-processing databases or web services that handle concurrent requests from different users. There may be performance benefits to using hardware techniques that exploit nested speculative parallelism within each coarse-grained task while composing parallelism across concurrent coarse tasks to decouple virtual-time commits in hardware [58]. Future work can explore how to extend S5 to provide simple, general primitives to make it easier for users to exploit explicit concurrency. Further examining the interaction of nested task-level speculation with memory consistency models may be fruitful.

Hardware support for concurrency could also improve the implementation of S5’s abstract data types. For example, tasks that perform insertions into `s5::fifo` or `s5::bag` shards write data to memory that will not be consumed until some time after their enclosing Fractal domain finishes. Instead of having these tasks participate in the virtual-time commit protocol as ordinary tasks, future work could explore decoupling the commit of shard-insertion tasks so that they commit in order relative to each other but may commit out of order with respect to ordinary tasks in the same domain, alleviating commit queue pressure. Decoupling the commit protocol may improve the system’s latency tolerance and make scaling to distributed systems more practical.

As we add new primitives to support various computation patterns, an important question is whether we can design these diverse primitives to work well together. This includes both the hardware primitives used by compilers and low-level systems programmers and software primitives used by application programmers. It is desirable to preserve composability [86]—that is, enabling separately written subroutines that use different primitives to be composed together to form more complex programs in a modular fashion. It is also desirable to continue to exploit the performance benefits of parallel composition and ensure using one primitive does not impede overlapping the execution of subroutines that use other primitives.

Heterogeneous and specialized processing elements: There is also significant room for further optimization for power- and area-efficiency in our hardware designs, which currently use general-purpose programmable cores to execute all tasks. One approach to specialization is to identify what operations are heavily used in speculative parallel algorithms—such as frequent task spawns—and optimizing the microarchitecture to provide high throughput for common operations while reducing expenditure on other less-used microarchitectural features. Another approach would be to replace some cores entirely with specialized processing elements to execute common tasks. Careful evaluation of hardware prototypes, using either FPGA-based or ASIC designs, may elucidate what sort of hardware specialization provides the best performance benefits, as well as whether using simpler specialized cores enables adding more cores to a chip or increasing clock frequencies while staying within a power and area budget.

While S5 was evaluated on complex programs that required tasks to execute on general-purpose programmable cores, custom spills are simple

to implement and may be useful even in specialized accelerators. Future work can explore integrating custom spills in specialized accelerators that might lack general-purpose cores at all, such as fixed-function graph analytics accelerators based on Chronos [2]. While we executed S5’s custom spillers and requeuers on CPU cores, future work can explore how semi-custom spills can be implemented more efficiently with simpler specialized FSMs or reconfigurable elements.

Another promising direction is to investigate intermediate design points that retain general-purpose programmability while integrating heterogeneous hardware elements. If we add increasingly diverse processing elements that are specialized for different tasks, it may become expensive to replicate many types of elements on every hardware tile to preserve the homogeneity of hardware tiles. Our compilers currently exploit this homogeneity to simplify load balancing and data-centric scheduling. Without homogeneity, a tile may not be able to execute all tasks, or perhaps some tiles would fall back to executing tasks slowly on general-purpose cores if they lack a specialized processing element available on other tiles to accelerate that task. Tiles that process work at different rates may require more responsive load balancing techniques. For example, both Swarm’s generic spills and S5’s custom spills so far never requeue a task to any tile other than the one it spilled from, but perhaps a more complex runtime system could be developed to redistribute tasks leveraging custom spillers.

Specializing for other application domains: So far, two systems have been developed that specialize Swarm: Chronos focuses on graph analytics [2], while ASH so far only supports RTL simulations. But the techniques in these systems are general and could be applied to other workloads. Exploring additional workloads could open up additional discoveries in how to combine our techniques for speculative and dataflow execution with optimizations specific to new application domains.

ASH’s new design may also be directly useful for future work. Though we have focused on RTL simulation, our combination of dataflow and selective execution may be more broadly beneficial, both for other types of simulation (e.g., microarchitectural), and for workloads beyond simulation. Our compiler-based techniques may extend to these domains, making it possible to map their designs automatically.

More broadly, our open-source compiler components may be reusable. T4 is implemented within LLVM without many dependences on the Clang frontend nor on the x86_64 backend. It is possible to target

other architectures supported in recent versions of LLVM, such as RISC-V. The use of specialized RISC-V cores, often with custom ISA extensions, is increasingly used within specialized accelerators [65]. For new applications not written in C++, it is also easily feasible to bring T4's existing capabilities to other LLVM-compiled languages, such as Rust, Swift, Kotlin, Julia, Lua, and Haskell.

Bibliography

- [1] M. Abeydeera, “Scalable and broad hardware acceleration through practical speculative parallelism,” Ph.D. dissertation, Massachusetts Institute of Technology, 2021.
- [2] M. Abeydeera and D. Sanchez, “Chronos: Efficient speculative parallelism for accelerators,” in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020.
- [3] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez, “SAM: Optimizing multithreaded cores for speculative parallelism,” in *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-26)*, 2017.
- [4] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, “Unlocking concurrency: Multicore programming with transactional memory,” *ACM Queue*, vol. 4, no. 10, 2006.
- [5] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiawicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The MIT Alewife machine: architecture and performance,” *Proc. of the 22nd annual Intl. Symp. on Computer Architecture*, 1995.
- [6] I. Akturk and U. R. Karpuzcu, “AMNESIAC: Amnesic automatic computer,” in *Proc. of the 22nd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017.
- [7] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, “Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel,” in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018.

- [8] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, “Conversion of control dependence to data dependence,” in *Proc. of the 10th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 1983.
- [9] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967.
- [10] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “Opentuner: An extensible framework for program autotuning,” in *Proc. PACT*, 2014.
- [11] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, “Perspective: A sensible approach to speculative automatic parallelization,” in *Proc. of the 25th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020.
- [12] A. W. Appel, *Compiling with Continuations*. Cambridge university press, 1992.
- [13] A. W. Appel and D. B. MacQueen, “Standard ML of New Jersey,” in *Proc. PLILP*, 1991.
- [14] Arvind, “Passing the token,” in *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (ISCA-32)*, 2005.
- [15] Arvind, R. Nikhil, and K. Pingali, “I-Structures-Data Structures For Parallel Computing,” *ACM TOPLAS*, vol. 11, no. 4, 1989.
- [16] M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte, “Min-max heaps and generalized priority queues,” *Communications of the ACM*, vol. 29, no. 10, 1986.
- [17] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, “AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers,” in *Proc. of the 46th annual Intl. Symp. on Computer Architecture (ISCA-46)*, 2019.

- [18] J. Babb, R. Tessier, M. Dahl, S. Z. Hanono, D. M. Hoki, and A. Agarwal, “Logic emulation with virtual wires,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 6, 1997.
- [19] D. A. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2,” in *Proc. ICPP*, 2006.
- [20] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, “Mathematizing C++ concurrency,” in *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2011.
- [21] S. Beamer, “A case for accelerating software RTL simulation,” *IEEE Micro*, vol. 40, no. 4, pp. 112–119, 2020.
- [22] S. Beamer and D. Donofrio, “Efficiently exploiting low activity factors to accelerate RTL simulation,” in *Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.
- [23] C. J. Beckmann and C. D. Polychronopoulos, “Fast barrier synchronization hardware,” in *Proc. of the ACM/IEEE conf. on Supercomputing (SC90)*, 1990.
- [24] D. K. Beece, G. Deiberg, G. Papp, and F. Villante, “The IBM engineering verification engine,” in *Proc. of the 25th Design Automation Conf. (DAC-25)*, 1988.
- [25] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications,” in *Proc. of the 9th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.
- [26] A. J. Bernstein, “Analysis of programs for parallel processing,” *IEEE Trans. Electronic Computers*, vol. EC-15, no. 5, 1966.
- [27] R. Bhagwan and B. Lin, “Fast and scalable priority queue architecture for high-speed network switches,” in *Proc. of the IEEE Infocom 2000*, 2000.

- [28] A. Bhowmik and M. Franklin, “A general compiler framework for speculative multithreading,” in *Proc. of the 14th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2002.
- [29] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, “Automatic intra-register vectorization for the Intel Architecture,” *International J. Parallel Programming*, vol. 30, no. 2, 2002.
- [30] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, “Internally deterministic parallel algorithms can be fast,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2012.
- [31] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence, “Parallel programming with Polaris,” *Computer*, vol. 29, no. 12, 1996.
- [32] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral program optimization system,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2008.
- [33] A. Bracy, K. Doshi, and Q. Jacobson, “Disintermediated active communication,” *Computer Architecture Letters*, vol. 5, no. 2, 2006.
- [34] A. Brahmakshatriya, E. Furst, V. Ying, C. Hsu, C. Hong, M. Rutenberberg, Y. Zhang, T. Jung, D. Richmond, M. Taylor, J. Shun, M. Oskin, D. Sanchez, and S. Amarasinghe, “Taming the zoo: A unified graph compiler framework for novel architectures,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (ISCA-48)*, 2021.
- [35] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, “Compiling graph applications for gpu with graphit,” in *Proc. of the 19th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2021.
- [36] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “The anatomy of the register file in a multiscalar processor,” in *Proc. of the 27th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-27)*, 1994.

- [37] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, “Revisiting the sequential programming model for multi-core,” in *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-40)*, 2007.
- [38] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, “TAO: Facebook’s distributed data store for the social graph,” in *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)*, 2013.
- [39] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous circuits*. Springer, 1995.
- [40] M. Budiu, P. Artigas, and S. Goldstein, “Dataflow: A complement to superscalar,” in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2005.
- [41] Cadence, “Palladium Z1 enterprise emulation platform,” https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/system-design-verification/palladium-z1-ds.pdf, 2015.
- [42] Cadence, “Protium X1 enterprise prototyping platform,” https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html, 2019.
- [43] 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. of the 40th annual Intl. Symp. on Computer Architecture (ISCA-40)*, 2013.
- [44] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, G.-Y. Wei, and D. Brooks, “HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs,” in *Proc. of the 41st annual Intl. Symp. on Computer Architecture (ISCA-41)*, 2014.
- [45] J. L. Carter and M. Wegman, “Universal classes of hash functions (extended abstract),” in *Proc. STOC-9*, 1977.
- [46] 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. of the 24th IEEE intl. symp. on High Performance Computer Architecture (HPCA-24)*, 2018.
- [47] K. M. Chandy and J. Misra, “Asynchronous distributed simulation via a sequence of parallel computations,” *Communications of the ACM*, vol. 24, no. 4, 1981.
 - [48] P. Chen, M. Hung, Y. Hwang, R. D. Ju, and J. K. Lee, “Compiler support for speculative multithreading architecture with probabilistic points-to analysis,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2003.
 - [49] V. Dadu, S. Liu, and T. Nowatzki, “PolyGraph: Exposing the value of flexibility for graph processing accelerators,” in *Proc. of the 48th annual Intl. Symp. on Computer Architecture (ISCA-48)*, 2021.
 - [50] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, “Merrimac: Supercomputing with streams,” in *Proc. of the 17th annual Intl. Conf. on Supercomputing*, 2003.
 - [51] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer, “RustBelt meets relaxed memory,” in *Proc. of the 46th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2019.
 - [52] 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.
 - [53] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, “Unconventional parallelization of nondeterministic applications,” in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018.
 - [54] C. Demetrescu, A. Goldberg, and D. Johnson, “9th DIMACS implementation challenge: Shortest paths,” <http://www.dis.uniroma1.it/~challenge9>, archived at <https://perma.cc/5KYT-YM36>, 2006.

- [55] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *Proc. of the 2nd annual Intl. Symp. on Computer Architecture (ISCA-2)*, 1975.
- [56] L. Dhulipala, G. Blelloch, and J. Shun, “Julienne: A framework for parallel graph algorithms using work-efficient bucketing,” in *Proc. of the 29th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2017.
- [57] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai, “A cost-driven compilation framework for speculative parallelization of sequential programs,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2004.
- [58] R. B. Durfee, “Nexion: Enabling concurrency on architectures for ordered parallelism,” Master’s thesis, Massachusetts Institute of Technology, 2022.
- [59] T. J. K. Edler von Koch and B. Franke, “Limits of region-based dynamic binary parallelization,” in *Proc. VEE*, 2013.
- [60] C. Eksombatchai, P. Jindal, J. Z. Liu, Y. Liu, R. Sharma, C. Sugnet, M. Ulrich, and J. Leskovec, “Pixie: A system for recommending 3+ billion items to 200+ million users in real-time,” in *Proceedings of the 2018 World Wide Web Conference*, 2018.
- [61] S. Eldridge, “Simulation speed for rocket-chip: VCS vs Verilator,” <https://github.com/chipsalliance/rocket-chip/issues/2160>, 2019.
- [62] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proc. of the 38th annual Intl. Symp. on Computer Architecture (ISCA-38)*, 2011.
- [63] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, “A survey on thread-level speculation techniques,” *ACM CSUR*, vol. 49, no. 2, 2016.
- [64] 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. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-43)*, 2010.

- [65] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, J. Shajrawi, K. Quinn, N. Sreedhara, P. Kansal, W. Wei, D. Jayaraman, L. Cheng, P. Chopda, E. Wang, A. Bikumandla, A. Karthik Sengottuvel, K. Thottempudi, A. Narasimha, B. Dodds, C. Gao, J. Zhang, M. Al-Sanabani, A. Zehtabioskuie, J. Fix, H. Yu, R. Li, K. Gondkar, J. Montgomery, M. Tsai, S. Dwarakapuram, S. Desai, N. Avidan, P. Ramani, K. Narayanan, A. Mathews, S. Gopal, M. Naumov, V. Rao, K. Noru, H. Reddy, P. Venkatapuram, and A. Bjorlin, “MTIA: First generation silicon targeting Meta’s recommendation systems,” in *Proc. of the 50th annual Intl. Symp. on Computer Architecture (ISCA-50)*, 2023.
- [66] J. Fix, N. P. Nagendra, S. Apostolakis, H. Zhang, S. Qiu, and D. I. August, “Hardware multithreaded transactions,” in *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018.
- [67] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the Cilk-5 multithreaded language,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1998.
- [68] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan, “Design and evaluation of the rollback chip: special purpose hardware for Time Warp,” *IEEE Trans. Computers*, vol. 41, no. 1, 1992.
- [69] R. Fujimoto, “The virtual time machine,” in *Proc. of the 1st ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 1989.
- [70] R. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, vol. 33, no. 10, 1990.
- [71] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh, “CEDAR: A large scale multiprocessor,” *SIGARCH Comput. Archit. News*, vol. 11, no. 1, 1983.
- [72] 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.*

of the 9th IEEE intl. symp. on High Performance Computer Architecture (HPCA-9), 2003.

- [73] S. Ghemawat and P. Menage, “TCMalloc: Thread-caching malloc,” <https://gperftools.github.io/gperftools/tcmalloc.html>, archived at <https://perma.cc/EK9E-LBYU>, 2007.
- [74] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, “Abelian: A compiler for graph analytics on distributed, heterogeneous platforms,” in *Euro-Par*, 2018.
- [75] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, “RipTide: A programmable, energy-minimal dataflow compiler and architecture,” in *Proc. of the 55th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-55)*, 2022.
- [76] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison-Wesley, 2006.
- [77] S. C. Goldstein, K. E. Schauser, and D. E. Culler, “Lazy threads: Implementing a fast parallel call,” *J. Parallel and Distributed Computing*, vol. 37, no. 1, 2006.
- [78] G. Graunke and S. Thakkar, “Synchronization algorithms for shared-memory multiprocessors,” *Computer*, vol. 23, no. 6, 1990.
- [79] 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.
- [80] S. Grossman, H. Litz, and C. Kozyrakis, “Making pull-based graph processing performant,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2018.
- [81] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Communications of the ACM*, vol. 31, no. 5, 1988.
- [82] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D’Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, “Haswell: The fourth-generation Intel Core processor,” *IEEE Micro*, vol. 34, no. 2, 2014.

- [83] L. Hammond, M. Willey, and K. Olukotun, “Data speculation support for a chip multiprocessor,” in *Proc. of the 8th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.
- [84] 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. of the 31st annual Intl. Symp. on Computer Architecture (ISCA-31)*, 2004.
- [85] H. Han and C.-W. Tseng, “Improving compiler and run-time support for irregular reductions using local writes,” in *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1998.
- [86] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, “Composable memory transactions,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2005.
- [87] M. A. Hassaan, M. Burtscher, and K. Pingali, “Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2011.
- [88] A. Heidarshenas, T. Gangwani, S. Yesil, A. Morrison, and J. Torrellas, “Snug: Architectural support for relaxed concurrent priority queueing in chip multiprocessors,” in *Proc. of the Intl. Conf. on Supercomputing (ICS’20)*, 2020.
- [89] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.
- [90] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*, 1993.
- [91] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art of Multiprocessor Programming*, 2nd ed. Morgan Kaufmann, 2020.
- [92] M. D. Hill and M. Marty, “Amdahl’s law in the multicore era,” *Computer*, vol. 41, no. 7, 2008.

- [93] M. Hill, “Multiprocessors should support simple memory consistency models,” *Computer*, vol. 31, no. 8, 1998.
- [94] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?” in *Proc. PASTE*, 2001.
- [95] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, “MultiGraph: Efficient graph processing on GPUs,” in *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-26)*, 2017.
- [96] J. Huang, P. Prabhu, T. B. Jablin, S. Ghosh, S. Apostolakis, J. W. Lee, and D. I. August, “Speculatively exploiting cross-invocation parallelism,” in *Proc. of the 25th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-25)*, 2016.
- [97] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, “Decoupled software pipelining creates parallelization opportunities,” in *Proc. of the 8th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2010.
- [98] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg, “McRT-Malloc: A scalable transactional memory allocator,” in *Proc. ISMM*, 2006.
- [99] W. N. Hung and R. Sun, “Challenges in large FPGA-based logic emulation systems,” in *Proceedings of the 2018 International Symposium on Physical Design*, 2018.
- [100] IBS data on IC design costs, “As chip design costs skyrocket, 3nm process node is in jeopardy,” <https://www.extremetech.com/computing/272096-3nm-process-node>, 2018.
- [101] A. Ioannou and M. Katevenis, “Pipelined heap (priority queue) management for advanced scheduling in high-speed networks,” in *International Conference on Communications (ICC)*, 2001.
- [102] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar, “Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies,” in *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013.

- [103] J. Janssen and H. Corporaal, “Making graphs reducible with controlled node splitting,” *ACM TOPLAS*, vol. 19, no. 6, 1997.
- [104] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Dilorieto, “Time warp operating system,” in *Proc. of the 11st Symp. on Operating System Principles (SOSP-11)*, 1987.
- [105] D. R. Jefferson, “Virtual time,” *ACM TOPLAS*, vol. 7, no. 3, 1985.
- [106] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, “Data-centric execution of speculative parallel programs,” in *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-49)*, 2016.
- [107] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, “A scalable architecture for ordered parallelism,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*, 2015.
- [108] 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.
- [109] 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. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*, 2018.
- [110] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. Martinez Caamaño, “Dynamic and speculative polyhedral parallelization using compiler-generated skeletons,” *International J. Parallel Programming*, vol. 42, no. 4, 2014.
- [111] N. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, “Speculative separation for privatization and reductions,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.
- [112] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, “Min-cut program decomposition for thread-level speculation,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2004.

- [113] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, “TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proc. of the 50th annual Intl. Symp. on Computer Architecture (ISCA-50)*, 2023.
- [114] V. Kanvar and U. P. Khedker, “Heap abstractions for static analysis,” *ACM CSUR*, vol. 49, no. 2, 2016.
- [115] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, “FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud,” in *Proc. of the 45th annual Intl. Symp. on Computer Architecture (ISCA-45)*, 2018.
- [116] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, 1998.
- [117] A. Kejariwal, X. Tian, M. Girkar, W. Li, S. Kozhukhov, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos, “Tight analysis of the performance potential of thread speculation using SPEC CPU2006,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2007.
- [118] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. Patel, “Rigel: An architecture and scalable programming interface for a 1000-core accelerator,” in *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.
- [119] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, “Imagine: media processing with streams,” *Micro, IEEE*, vol. 21, no. 2, 2001.
- [120] U. R. Khan, H. L. Owen, and J. L. Hughes, “FPGA architectures for ASIC hardware emulators,” in *Sixth Annual IEEE International ASIC Conference and Exhibit*, 1993.
- [121] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *Proc.*

- of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), 2002.
- [122] D. Kim, J. Zhao, J. Bachrach, and K. Asanović, “Simmani: Runtime power modeling for arbitrary RTL with automatic signal selection,” in *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-52)*, 2019.
 - [123] S. Kim, J. Kim, M. J. Kim, W. Jung, J. Kim, M. Rhu, and J. H. Ahn, “BTS: An accelerator for bootstrappable fully homomorphic encryption,” in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49)*, 2022.
 - [124] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz, “RAMP Blue: A message-passing manycore system in FPGAs,” in *2007 International Conference on Field Programmable Logic and Applications*, 2007.
 - [125] V. Krishnan and J. Torrellas, “The need for fast communication in hardware-based speculative chip multiprocessors,” in *Proc. of the 8th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-8)*, 1999.
 - [126] H. Krupnova and G. Saucier, “FPGA-based emulation: Industrial and custom prototyping solutions,” in *Proc. of the 10th intl. conf. on Field-Programmable Logic and Applications (FPL)*, 2000.
 - [127] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, “Optimistic parallelism requires abstractions,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2007.
 - [128] S. Kumar, C. Hughes, and A. Nguyen, “Carbon: Architectural support for fine-grained parallelism on chip multiprocessors,” in *Proc. of the 34th annual Intl. Symp. on Computer Architecture (ISCA-34)*, 2007.
 - [129] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, “Repairing sequential consistency in C/C++11,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2017.
 - [130] W. Landi, “Undecidability of static analysis,” *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, 1992.

- [131] S. Larsen, E. Witchel, and S. Amarasinghe, “Increasing and detecting memory address congruence,” in *Proc. of the 11st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-11)*, 2002.
- [132] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Proc. of the 2nd IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2004.
- [133] D. Lea, “JSR-166, the proposed Java Concurrency Package: Hash table util.concurrent.ConcurrentHashMap,” *JSRs: Java Specification Requests*, 2003.
- [134] E. A. Lee, “The problem with threads,” *Computer*, vol. 39, no. 5, 2006.
- [135] C. Leiserson and T. Schardl, “A work-efficient parallel breadth-first search algorithm,” in *Proc. of the 22nd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2010.
- [136] J. Leskovec and A. Krevl, “SNAP datasets: Stanford large network dataset collection,” <http://snap.stanford.edu/data>, 2014.
- [137] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters,” *Internet Mathematics*, vol. 6, no. 1, 2009.
- [138] M. Lipasti and J. Shen, “Exceeding the dataflow limit via value prediction,” in *Proc. of the 29th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-29)*, 1996.
- [139] J. D. C. Little, “A proof for the queueing formula: $L = \lambda W$,” *Operations Research*, vol. 9, no. 3, 1961.
- [140] Q. C. Liu, J. Shi, S. Yu, L. Dhulipala, and J. Shun, “Parallel batch-dynamic algorithms for k-core decomposition and related graph problems,” in *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, 2022.
- [141] S. Liu, G. Bai, J. Sun, and J. S. Dong, “Towards using concurrent Java API correctly,” in *21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2016.

- [142] 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. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2006.
- [143] 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. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.
- [144] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, “Challenges in parallel graph processing,” *Parallel Processing Letters*, 2007.
- [145] K. Mackenzie, J. Kubiawicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. Kaashoek, “Exploiting two-case delivery for fast protected messaging,” in *Proc. of the 4th intl. symp. on High-Performance Computer Architecture*, 1998.
- [146] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, “An evaluation of vectorizing compilers,” in *Proc. of the 20th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-20)*, 2011.
- [147] J. F. Martínez and J. Torrellas, “Speculative synchronization: Applying thread-level speculation to explicitly parallel applications,” in *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.
- [148] J. M. Martínez Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, “APOLLO: Automatic speculative POLyhedral Loop Optimizer,” in *Proc. IMPACT*, 2017.
- [149] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam, “Array-data flow analysis and its use in array privatization,” in *Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 1993.
- [150] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. Computer Systems*, vol. 9, no. 1, 1991.

- [151] Mentor/Siemens, “Veloce Strato,” <https://eda.sw.siemens.com/en-US/ic/veloce/strato-hardware/>, 2017.
- [152] Microsoft, “ConcurrentBag<T> Class,” <https://learn.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1?view=netframework-4.0>, archived at <https://perma.cc/Z4N2-NXLD>, 2010.
- [153] Microsoft, “What’s new in the .NET Framework 4,” [https://learn.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms171868\(v=vs.100\)](https://learn.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/ms171868(v=vs.100)), archived at <https://perma.cc/9RNY-S4TC>, 2010.
- [154] S. P. Midkiff, “Automatic parallelization: An overview of fundamental compiler techniques,” *Synthesis Lectures on Computer Architecture*, 2012.
- [155] M. Moir and N. Shavit, “Concurrent data structures (book chapter),” in *Handbook of Data Structures and Applications*, D. Methta and S. Sahni, Eds., 2004.
- [156] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “LogTM: Log-based transactional memory,” in *Proc. of the 12nd IEEE intl. symp. on High Performance Computer Architecture (HPCA-12)*, 2006.
- [157] 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. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-52)*, 2019.
- [158] 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. of the 12nd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.
- [159] V. Nagarajan and R. Gupta, “ECMon: exposing cache events for monitoring,” in *Proc. of the 36th annual Intl. Symp. on Computer Architecture*, 2009.
- [160] Nangate Inc., “The NanGate 45nm Open Cell Library,” http://www.nangate.com/?page_id=2325, 2008.

- [161] Q. M. Nguyen and D. Sanchez, “Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism,” in *Proc. of the 53rd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-53)*, 2020.
- [162] Q. M. Nguyen and D. Sanchez, “Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism,” in *Proc. of the 29th IEEE intl. symp. on High Performance Computer Architecture (HPCA-29)*, 2023.
- [163] R. Nikhil and Arvind, “Executing a program on the MIT tagged-token dataflow architecture,” *IEEE Trans. Computers*, vol. 39, no. 3, 1990.
- [164] M. Noakes, D. Wallach, and W. Dally, “The J-Machine multicomputer: An architectural evaluation,” in *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*, 1993.
- [165] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, “Stream-dataflow acceleration,” in *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*, 2017.
- [166] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, “Exploring the potential of heterogeneous Von Neumann/dataflow execution models,” in *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (ISCA-42)*, 2015.
- [167] D. J. Nutile, “Using profiling to improve the performance of automatically parallelized programs,” Master’s thesis, Massachusetts Institute of Technology, 2019.
- [168] J. Oh, M. Prvulovic, and A. Zajic, “TLSync: support for multiple fast barriers using on-chip transmission lines,” in *Proc. ISCA*, 2011.
- [169] J. T. Oplinger, D. L. Heine, and M. S. Lam, “In search of speculative thread-level parallelism,” in *Proc. of the 8th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-8)*, 1999.
- [170] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, “Automatic thread extraction with decoupled software pipelining,” in *Proc. of the 38th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-38)*, 2005.

- [171] V. Packirisamy, A. Zhai, W.-C. Hsu, P. C. Yew, and T. F. Ngai, “Exploring speculative parallelism in SPEC2006,” in *Proc. of the IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2009.
- [172] S. Pallottino and M. G. Scutellà, “Shortest path algorithms in transportation models: Classical and innovative aspects,” in *Equilibrium and Advanced Transportation Modelling*, 1998.
- [173] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, “Controlling program execution through binary instrumentation,” *SIGARCH Computer Architecture News*, vol. 33, no. 5, 2005.
- [174] C. H. Papadimitriou, “The serializability of concurrent database updates,” *Journal of the ACM (JACM)*, vol. 26, no. 4, 1979.
- [175] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, “Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation,” in *Proc. of the 37th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-37)*, 2004.
- [176] Y. N. Patt, W.-m. Hwu, and M. Shebanow, “HPS, a new microarchitecture: Rationale and introduction,” in *Proc. of the 18th annual workshop and symp. on Microprogramming and Microarchitecture (MICRO-18)*, 1985.
- [177] G. A. Pavlopoulos, P. I. Kontou, A. Pavlopoulou, C. Bouyioukos, E. Markou, and P. G. Bagos, “Bipartite graphs in systems biology and medicine: a survey of methods and applications,” *GigaScience*, vol. 7, no. 4, 2018.
- [178] Z. Peng, A. Powell, B. Wu, T. Bicer, and B. Ren, “Graphphi: Efficient parallel graph processing on emerging throughput-oriented architectures,” in *Proc. of the 27th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-27)*, 2018.
- [179] G. F. Pfister, “The Yorktown simulation engine: Introduction,” in *Proc. of the 19th Design Automation Conf. (DAC-19)*, 1982.
- [180] 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. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.
- [181] C. Pit-Claudel, T. Bourgeat, S. Lau, Arvind, and A. Chlipala, “Effective simulation and debugging for a high-level hardware language using software compilers,” in *Proc. of the 26th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, 2021.
 - [182] L. Porter, B. Choi, and D. Tullsen, “Mapping out a path from hardware transactional memory to speculative multithreading,” in *Proc. of the 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-18)*, 2009.
 - [183] G. Posluns, Y. Zhu, G. Zhang, and M. C. Jeffrey, “A scalable architecture for reprioritizing ordered parallelism,” in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49)*, 2022.
 - [184] P. Prabhu, G. Ramalingam, and K. Vaswani, “Safe programmable speculative parallelism,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2010.
 - [185] M. Prvulovic, M. Garzaran, L. Rauchwerger, and J. Torrellas, “Removing architectural bottlenecks to the scalability of speculative parallelization,” in *Proc. of the 28th annual Intl. Symp. on Computer Architecture (ISCA-28)*, 2001.
 - [186] X. Qian, B. Sahelices, and J. Torrellas, “OmniOrder: Directory-based conflict serialization of transactions,” in *Proc. of the 41st annual Intl. Symp. on Computer Architecture (ISCA-41)*, 2014.
 - [187] 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. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.
 - [188] H. E. Ramadan, C. J. Rossbach, and E. Witchel, “Dependence-aware transactional memory for increased concurrency,” in *Proc. of the 41st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-41)*, 2008.

- [189] G. Ramalingam, “The undecidability of aliasing,” *ACM Trans. Programming Languages and Systems*, vol. 16, no. 5, 1994.
- [190] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, “Speculative parallelization using software multi-threaded transactions,” in *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010.
- [191] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, “Decoupled software pipelining with the synchronization array,” in *Proc. of the 13rd Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-13)*, 2004.
- [192] L. Rauchwerger and D. Padua, “The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization,” in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1995.
- [193] 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. of the Intl. Conf. on Supercomputing (ICS’05)*, 2005.
- [194] 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. of the Intl. Conf. on Supercomputing (ICS’05)*, 2005.
- [195] L. Rizzatti, “Hardware emulation: Three decades of evolution. part III,” *Verification Horizons*, pp. 15–18, 2015.
- [196] L. Rizzatti, “Hardware emulation: Three decades of evolution. part I,” *Verification Horizons*, pp. 26–27, 2015, https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-11_issue-1/articles/stream/hardware-emulation-three-decades-of-evolution_vh-v11-i1.pdf.
- [197] L. Rizzatti, “Hardware emulation: Three decades of evolution. part II,” *Verification Horizons*, pp. 40–42, 2015, https://s3.amazonaws.com/verificationhorizons.verificationacademy.com/volume-11_issue-2/articles/stream/hardware-emulation-three-decades-of-evolution-part-II_vh-v11-i2.pdf.

- [198] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *Proc. AAAI-29*, 2015.
- [199] E. Rotem, Y. Mandelblat, V. Basin, E. Weissmann, A. Gihon, R. Chabukswar, R. Fenger, and M. Gupta, “Alder Lake Architecture,” in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.
- [200] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, “An architecture of a dataflow single chip processor,” in *Proc. of the 16th annual Intl. Symp. on Computer Architecture (ISCA-16)*, 1989.
- [201] J. Salamanca, J. N. Amaral, and G. Araujo, “Evaluating and improving thread-level speculation in hardware transactional memories,” in *Proc. IPDPS*, 2016.
- [202] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, “F1: A fast and programmable accelerator for fully homomorphic encryption,” in *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-54)*, 2021.
- [203] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, and D. Sanchez, “CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data,” in *Proc. of the 49th annual Intl. Symp. on Computer Architecture (ISCA-49)*, 2022.
- [204] D. Sanchez, R. Yoo, and C. Kozyrakis, “Flexible architectural support for fine-grain scheduling,” in *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010.
- [205] A. E. Sariyüce, C. Seshadhri, A. Pinar, and U. V. Çatalyürek, “Nucleus decompositions for identifying hierarchy of dense subgraphs,” *ACM Trans. Web*, vol. 11, no. 3, 2017.
- [206] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding fork-join parallelism into LLVM’s intermediate representation,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

- [207] S. L. Scott, “Synchronization and communication in the T3E multiprocessor,” in *Proc. of the 7th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.
- [208] S. B. Seidman, “Network structure and minimum degree,” *Social Networks*, vol. 5, no. 3, 1983.
- [209] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, “GraphJet: Real-time content recommendations at Twitter,” in *Proc. of the VLDB Endowment (VLDB)*, 2016.
- [210] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.
- [211] J. Shi, L. Dhulipala, and J. Shun, “Parallel clique counting and peeling algorithms,” in *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, 2021.
- [212] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2013.
- [213] J. Shun, L. Dhulipala, and G. E. Blelloch, “Smaller and faster: Parallel processing of compressed graphs with ligra+,” in *2015 Data Compression Conference*, 2015.
- [214] J. E. Smith and A. R. Pleszkun, “Implementation of precise interrupts in pipelined processors,” in *Proc. of the 12nd annual Intl. Symp. on Computer Architecture (ISCA-12)*, 1985.
- [215] W. Snyder, “Verilator,” <https://www.veripool.org/verilator/>, 2003.
- [216] W. Snyder, “Verilator 4.0: Open source simulation goes multi-threaded,” in *The Open Source Digital Design Conference (OR-Conf)*, 2018.
- [217] W. Snyder, “Verilator, accelerated,” in *2nd Workshop on Open-Source Design Automation (OSDA)*, 2020.

- [218] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, “Multiscalar processors,” in *Proc. of the 22nd annual Intl. Symp. on Computer Architecture (ISCA-22)*, 1995.
- [219] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott, “Alert-on-update: a communication aid for shared memory multiprocessors,” in *Proc. of the 12th ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*, 2007.
- [220] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally, “Evaluation of mechanisms for fine-grained parallel programs in the J-machine and the CM-5,” in *Proc. of the 20th annual Intl. Symp. on Computer Architecture (ISCA-20)*, 1993.
- [221] SpinalHDL, “A FPGA friendly 32 bit RISC-V CPU implementation,” <https://github.com/SpinalHDL/VexRiscv>, 2018.
- [222] G. L. Steele Jr, “RABBIT: A compiler for SCHEME,” Massachusetts Institute of Technology, Tech. Rep. AITR-474, 1978.
- [223] G. L. Steele Jr and G. J. Sussman, “Lambda: The ultimate imperative,” Massachusetts Institute of Technology, Tech. Rep. AIM-353, 1976.
- [224] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, “The STAMPede approach to thread-level speculation,” *ACM TOCS*, vol. 23, no. 3, 2005.
- [225] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, “A scalable approach to thread-level speculation,” in *Proc. of the 27th annual Intl. Symp. on Computer Architecture (ISCA-27)*, 2000.
- [226] J. G. Steffan and T. C. Mowry, “The potential for using thread-level data speculation to facilitate automatic parallelization,” in *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture (HPCA-4)*, 1998.
- [227] 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. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*, 2017.

- [228] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, “Graphgrind: Addressing load imbalance of graph partitioning,” in *Proc. of the Intl. Conf. on Supercomputing (ICS’18)*, 2018.
- [229] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, “Graphmat: High performance graph analytics made productive,” in *Proc. of the VLDB Endowment (VLDB)*, 2015.
- [230] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, “The WaveScalar architecture,” *ACM Trans. Computer Systems (TOCS)*, vol. 25, no. 2, 2007.
- [231] Swarm Architecture Team, “The swarm architecture simulator,” <https://github.com/SwarmArch/sim>, 2021.
- [232] Synopsys Inc., “ZeBu Server 4,” <https://www.synopsys.com/verification/emulation/zebu-server.html>, 2018.
- [233] Z. Tan, Z. Qian, X. Chen, K. Asanovic, and D. Patterson, “DIA-BLO: A warehouse-scale computer network simulator using FPGAs,” in *Proc. of the 20th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, 2015.
- [234] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, “RAMP Gold: an FPGA-based architecture simulator for multiprocessors,” in *Proc. of the 47th Design Automation Conf. (DAC-47)*, 2010.
- [235] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, “A case for FAME: FPGA architecture model execution,” in *Proc. of the 37th annual Intl. Symp. on Computer Architecture (ISCA-37)*, 2010.
- [236] R. E. Tarjan, “Edge-disjoint spanning trees and depth-first search,” *Acta Informatica*, vol. 2, no. 6, 1976.
- [237] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal, “The Raw microprocessor: a

- computational fabric for software circuits and general-purpose programs,” in *Proc. of the 35th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-35)*, 2002.
- [238] The Chronos FPGA Framework to accelerate ordered applications, “<https://github.com/SwarmArch/chronos/>,” 2020.
 - [239] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, “Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics,” in *Proc. of the 54th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-54)*, 2021.
 - [240] R. M. Tomasulo, “An efficient algorithm for exploiting multiple arithmetic units,” *IBM Journal of Research and Development*, vol. 11, no. 1, 1967.
 - [241] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, “TriCheck: Memory model verification at the tri-section of software, hardware, and isa,” in *Proc. of the 22nd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017.
 - [242] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, “Compiler techniques for the superthreaded architectures,” *International J. Parallel Programming*, vol. 27, no. 1, 1999.
 - [243] R. Turner, “A primer on processor-based emulation,” EE-Times, <https://www.eetimes.com/a-primer-on-processor-based-emulation/>, 2004.
 - [244] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, “Speculative decoupled software pipelining,” in *Proc. of the 16th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-16)*, 2007.
 - [245] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, 1990.
 - [246] R. A. van Engelen, “Efficient symbolic analysis for optimizing compilers,” in *Proc. CC*, 2001.
 - [247] H. Vandierendonck, S. Rul, and K. De Bosschere, “The Paralax infrastructure: Automatic parallelization with a helping hand,”

- in *Proc. of the 19th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-19)*, 2010.
- [248] T. Vijaykumar and G. S. Sohi, “Task selection for a multiscalar processor,” in *Proc. of the 31st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-31)*, 1998.
 - [249] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, “Active messages: a mechanism for integrated communication and computation,” in *Proc. of the 19th annual Intl. Symp. on Computer Architecture (ISCA-19)*, 1992.
 - [250] H. Wang and S. Beamer, “RepCut: Superlinear parallel RTL simulation with replication-aided partitioning,” in *Proc. of the 28th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, 2023.
 - [251] 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.
 - [252] J. Whaley and C. Kozyrakis, “Heuristics for profile-driven method-level speculative parallelization,” in *Proc. ICPP*, 2005.
 - [253] C. Wolf, “Yosys Open SYNthesis Suite,” <http://www.clifford.at/yosys/>, 2014.
 - [254] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang, “Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor,” in *Proc. of the 17th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.
 - [255] Xilinx, “Alveo U250 data center accelerator card,” <https://www.xilinx.com/products/boards-and-kits/alveo/u250.html>, 2019.
 - [256] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, “Hybrid dataflow/von-Neumann architectures,” *IEEE Trans. Parallel and Distributed Systems*, 2013.
 - [257] 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. of the 13rd IEEE intl. symp. on High Performance Computer Architecture (HPCA-13)*, 2007.
- [258] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph convolutional neural networks for web-scale recommender systems,” in *Proc. of the 24th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2018.
 - [259] V. A. Ying, “Swarm user guide,” <https://github.com/SwarmArch/sim/blob/main/doc/UserGuide.mdown#low-level-swarm-programming>, archived at <https://perma.cc/WH6K-PTW7>, 2021.
 - [260] V. A. Ying, M. C. Jeffrey, and D. Sanchez, “T4: Compiling sequential code for effective speculative parallelization in hardware,” in *Proc. of the 47th annual Intl. Symp. on Computer Architecture (ISCA-47)*, 2020.
 - [261] T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, and S. Sakai, “Dataflow computer development in Japan,” in *Proc. of the Intl. Conf. on Supercomputing (ICS’90)*, 1990.
 - [262] R. Zahir, “Medfield smartphone SOC Intel Atom Z2460 processor,” in *2012 IEEE Hot Chips 24 Symposium (HCS)*, 2012.
 - [263] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, “Compiler optimization of scalar value communication between speculative threads,” in *Proc. of the 10th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.
 - [264] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, “Compiler optimization of memory-resident value communication between speculative threads,” in *Proc. of the 2nd IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2004.
 - [265] G. Zhang, V. Chiu, and D. Sanchez, “Exploiting semantic commutativity in hardware speculation,” in *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-49)*, 2016.
 - [266] G. Zhang, W. Horn, and D. Sanchez, “Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent

- systems,” in *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*, 2015.
- [267] G. Zhang and D. Sanchez, “Leveraging caches to accelerate hash tables and memoization,” in *Proc. of the 52nd annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-52)*, 2019.
 - [268] Y. Zhang and E. A. Hansen, “Parallel breadth-first heuristic search on a shared-memory architecture,” in *AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.
 - [269] Y. Zhang, L. Rauchwerger, and J. Torrellas, “Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors,” in *Proc. of the 4th IEEE intl. symp. on High Performance Computer Architecture (HPCA-4)*, 1998.
 - [270] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, “Optimizing ordered graph algorithms with GraphIt,” in *Proc. of the 18th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2020.
 - [271] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, “GraphIt: A high-performance graph DSL,” in *Proc. of the ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2018.
 - [272] C. Zilles and L. Baugh, “Extending hardware transactional memory to support non-busy waiting and non-transactional actions,” in *TRANSACT*, 2006.