Scaling Sequential Code with Hardware–Software Co-Design for Fine-Grain Speculative Parallelization

by

Victor A. Ying

B.S.E. in Electrical Engineering
Princeton University, 2016

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2019

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

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

Scaling Sequential Code with Hardware–Software Co-Design for Fine-Grain Speculative Parallelization

by

Victor A. Ying

Submitted to the Department of
Electrical Engineering and Computer Science
on May 23, 2019, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Multicores are now ubiquitous, but most programmers still write sequential code. Speculative parallelization is an enticing approach to parallelize code while retaining the ease and simplicity of sequential programming, making parallelism pervasive. However, prior speculative parallelizing compilers and architectures achieved limited speedups due to high costs of recovering from misspeculation, limited support for fine-grain parallelism, and hardware scalability bottlenecks.

We present SCC, a parallelizing compiler for sequential C/C++ programs. SCC targets the recent Swarm architecture, which exposes a flexible execution model, enables fine-grain speculative parallelism, supports locality and composition, and scales efficiently. SCC introduces novel compiler techniques to exploit Swarm's features and parallelize a broader range of applications than prior work. SCC performs whole-program fine-grain parallelization, breaking applications into many small tasks of tens of instructions each, and decouples the spawning of speculative tasks to enable cheap selective aborts. SCC exploits parallelism across function calls, loops, and loop nests; performs new transformations to expose more speculative parallelism enabled by Swarm's execution model; and exploits locality across fine-grain tasks. As a result, SCC speeds up seven SPEC CPU2006 benchmarks by gmean 6.7× and by up to 29× on 36 cores, over optimized serial code.

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

Acknowledgments

This work would be impossible except as a collaboration with several individuals, to whom I offer my deep gratitude. First and foremost is my research advisor, Prof. Daniel Sanchez, who patiently taught me how to conduct research in computer architecture. Daniel has been a consistent source of encouragement, and he has taught me to identify interesting research questions, and then focus on key issues that will make an impact.

In my time at MIT, have I had the great privilege to collaborate with Daniel, Prof. Joel Emer, Mark Jeffrey, Suvinay Subramanian, Maleen Abeydeera, and Ryan Lee on two publications that have extended the Swarm architecture with new capabilities. I have learned much of what I know about how to reason about the benefits and pitfalls of speculative execution, as well as how to present research work, from working with these great co-authors. One of these collaborations yielded the Fractal execution model that is foundational to my compiler project, which is the focus of this thesis. Mark and Daniel have continued to work with me on the compiler project for the last two years, and their support made it possible to overcome the many challenges this project faced. I also benefited from the mentorship of Tao B. Schardl in compiler engineering. All of this work also builds upon years of prior work done by my collaborators and others to develop the Swarm architecture.

I have also had the great luck to have collaborated with three great undergraduates: Elliott Forde, Domenic Nutile, and Grace Yin. All three contributed to the compiler project and have been an absolute pleasure to work with.

I am thankful also for the support of our research group. Po-An Tsai, Anurag Mukkara, Guowei Zhang, and Quan Nguyen have been kind and thoughtful friends to me during the last three years, and they have often given me useful feedback that has improved the work presented here. I look forward to the years to come as I work towards my Ph.D. alongside this group.

Finally, I owe thanks to the many individuals who helped me find my way to working in computer systems, and supported me in my pursuits. My family, friends, teachers, and co-workers have indelibly shaped my life by teaching me the persistence needed to pursue rewarding challenges.

Contents

Chapter 1

Introduction

Multicore processors are now dominant in computer systems, and the number of cores available on general-purpose systems continues to grow. However, most programmers still write sequential code by default. Concurrent programming remains a specialized skill due to its difficulty, and is subject to many pitfalls such as deadlocks, data races, and non-determinism. To address this situation, systems should exploit the *implicit parallelism* available in sequential programs by automatically parallelizing them to run across many cores.

Unfortunately, current compilers struggle to extract significant parallelism from most sequential programs [4]. Although some programs feature regular code regions that are easy to parallelize, many sequential programs are *irregular*, featuring data-dependent control flow and dynamically determined data flow (e.g., due to pointer aliasing). This stymies compile-time parallelization, as the compiler cannot statically determine which work is independent and thus can run in parallel.

Parallelizing entire irregular programs requires exploiting *speculative parallelism*, which combines compile-time and run-time techniques. With speculative parallelization, the compiler divides code into *tasks* that are likely to be independent. At run-time, 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 semantics.

Speculative execution is onerous in software, and its overheads often negate the benefits of extra parallelism [22, 23]. Therefore, speculative parallelization is best done

with hardware support for speculative execution. This creates a tight dependence between compiler and hardware architecture: the features that the architecture exposes shape the compiler's design and limit the types of parallelism it can exploit.

Prior compilers and architectures for speculative parallelization, also known as thread-level speculation (TLS) systems, have achieved limited speedups on real-world applications [14, 17, 21, 25, 46, 47, 50, 52, 57, 65]. This is because prior TLS architectures suffer from several issues: *(i)* resolving dependences by aborting all later tasks *en masse*, making aborts very expensive; *(ii)* restrictive execution models that prevent spawning tasks in arbitrary order, limiting the kinds of parallel patterns that can be expressed; *(iii)* lack of support for locality-aware execution, which hurts efficiency and limits scalability; and *(iv)* serial task spawn and commit mechanisms, whose overheads prevent exploiting fine-grain parallelism. These issues limit TLS compilers and constrain the parallelism that they can uncover.

Recent work has proposed the Swarm architecture for speculative parallelization [26, 27, 28, 55]. Swarm builds on prior TLS and hardware transactional memory (HTM) systems, and addresses the above issues (Chapter 2): Swarm *(i)* selectively aborts tasks when resolving dependences, *(ii)* has an expressive execution model based on timestamped tasks that allows conveying more parallel constructs to hardware; *(iii)* includes mechanisms to exploit locality [26] and nested parallelism [55]; and *(iv)* performs ordered speculation in a distributed, scalable way, with overheads low enough to scale fine-grain tasks of tens of instructions to hundreds of cores. However, using these features effectively required extensive manual program transformations. Prior work has not studied whether exploiting Swarm features can be automated by a parallelizing compiler, or whether Swarm can offer improved performance without significant changes to programming practices.

1.1 Contributions

This thesis presents *SCC*, the Swarm C/C++ Compiler. SCC leverages Swarm's execution model to parallelize more control- and data-flow patterns than prior TLS compilers, exploits nesting to reduce the whole program to fine-grain tasks of a few tens to hundreds of instructions, exploits locality, and spawns tasks far in advance.

As a result, SCC breaks complete applications into hundreds to thousands of task types, and scales several full applications to tens of cores. SCC introduces four key contributions over prior TLS compilers:

- *Whole-program fine-grain parallelization:* SCC breaks *the entire program* into fine-grain tasks that exploit Swarm's cheap, selective aborts (Section 3.1). SCC transforms function calls to avoid using a call stack and to minimize the data communicated among tasks (Section 3.2). Most task spawns require passing only a few values through registers, instead of a full thread context as in TLS systems. SCC leverages hardware support for nesting for composition: functions and translation units are compiled independently, yet fine-grain tasks from functions at different call depths all run in parallel.
- *Unrestricted out-of-order task spawn:* SCC tags every task with a timestamp to record program order. This enables aggressive code transformations that are free to spawn tasks far in advance and in *any* order. SCC exploits this by spawning many child tasks from a single parent task, exposing greater parallelism (Section 3.3).
- *Progressive loop expansion* is a novel loop transformation that speculatively spawns many loop iterations in parallel, rapidly filling the system with tasks from irregular loops, e.g., those with unknown termination conditions. This transformation exploits unrestricted out-of-order spawn to achieve a highly parallel tree of tasks, with an asymptotically shorter critical path than prior work (Section 3.4).
- *Locality-aware speculative execution:* SCC inserts code to compute the cache line addresses a task will access before the task is spawned, and communicates these *spatial hints* to hardware. This lets hardware run tasks that access the same data at the same chip tile to exploit locality (Section 3.5).

We implement SCC within the LLVM compiler framework. SCC transforms the entire program one function at a time, not relying on link-time optimization or interprocedural analysis. Thus, SCC compilation times stay proportional to code size.

We evaluate SCC with SPEC CPU2006 C/C++ programs (Chapter 4). Some benchmarks automatically parallelize well in their original forms, scaling by up to 29× on 36 cores, and up to 69× on 100 cores. Others require minimal source code

modifications. These modifications are simple and retain the sequential semantics of the program, allowing programmers to unlock parallelism without changing how they reason about program behavior. Overall, SCC achieves gmean 6.7× speedup on 36 cores over the original serial code compiled with `-O3`. These gains apply both to in-order cores and to out-of-order cores that aggressively exploit ILP.

These results show that, by leveraging hardware support, SCC can effectively parallelize of a wide range of code patterns, sidestepping the limitations and complexity of previous compilers. SCC also opens up further avenues on automatic and compiler-aided parallelization.

Chapter 2

Background and Motivation

Almost all programmers first learn to write *sequential* programs, in which the behavior of the program is specified by code statements that, conceptually, run one at a time, in a program-specified total order. This program order makes sequential programs easier to write and easier to understand. A sequential program maps naturally to serial execution on a single processor core.

However, most computer systems today have multicore processors. To utilize multiple cores, a program must be parallelized. The goal of parallelization is to reduce the amount of time a program takes to run, or to enable a program to perform more computation in the same amount of time, by dividing the computational work of an program into *tasks* that execute simultaneously on multiple cores.

Parallel executions must run the tasks that make up a program in an order consistent with the *dependences* among tasks. Two tasks have a *data dependence* if one task produces a data value that is needed as an input to the other. Two tasks have a *control dependence* if one task's computation determines whether or not the other task should execute at all. In order to execute programs in parallel while maintaining the ease of sequential programming, a system must identify *independent* tasks that can be run in parallel.

2.1 Related work

We will now review prior work that has sought to parallelize sequential programs, categorized by the approaches they used to identify independent tasks.

2.1.1 Non-speculative parallelizing compilers

Non-speculative parallelizing compilers divide sequential code into tasks that are guaranteed to be independent and can thus run in parallel. The key limitation of non-speculative compilers is that ensuring that two tasks are independent is often impossible at compile time. Polyhedral compilers [6, 18] can parallelize loops that operate on regular data structures, such as arrays or matrices. But many programs are *irregular*: they include data-dependent control flow, or they operate on pointer-based structures, such as trees or graphs, for which static analyses are ineffective. In addition, many programs span multiple translation units and libraries, so compilers have limited visibility into invoked code, impeding non-speculative parallelization.

For irregular programs, non-speculative parallelization techniques have focused on exploiting *fine-grain pipeline parallelism* in inner loops. DSWP [37, 44, 45] pins loop iteration fragments across cores to localize loop-carried dependences, and relies on hardware support for fine-grain inter-core communication. HELIX [9] implements efficient inter-thread communication in software, which suffices for some programs, and HELIX-RC [8] relies on hardware support to decouple value communication among threads to accelerate a broader set of benchmarks. Unlike SCC, these techniques cannot compose parallelism across nested loops or function calls. Moreover, they resort to conservative serial execution of any code outside of the loops they can parallelize.

2.1.2 Speculative parallelizing compilers

Some compilers leverage data-dependence speculation to parallelize a broader range of programs. They divide sequential code into tasks that are *likely* to be independent, and attempt to run these tasks in parallel, gambling that data dependences will be rare. Data dependences are detected at runtime, and the system checks if the execution yielded results consistent with the discovered data dependences. If not, the tasks are

in *conflict*, so some tasks are aborted (i.e., their effects are rolled back, restoring a previous version of the system state) and may be re-executed.

Though speculative execution can be done in software, the overheads of conflict detection, version management, and thread synchronization often overwhelm the benefits of parallelism [22]. As a result, software-only speculative parallelizing compilers are practical only in limited cases [36].

Instead, much prior work advocates for hardware support for speculation. Speculative execution is much more efficient in hardware because it reuses already-existing mechanisms (caches for version management and cache coherence for conflict detection). However, hardware support introduces a tight dependence between architecture and compiler.

2.1.3 Thread-level speculation (TLS)

TLS architectures provide hardware support for speculative parallelization of sequential programs [14, 17, 21, 46, 47, 50, 53, 65]. Unfortunately, TLS architectures suffer from several issues: *(i)* unselective aborts, making speculation expensive; *(ii)* restrictive execution models that only allows spawning the current task's immediate successor, limiting the parallel patterns that can be expressed; *(iii)* lack of support for locality-aware execution; and *(iv)* implementations based on centralized structures or serialized commits that scale poorly beyond a few cores, with overheads too high to exploit fine-grain parallelism. These issues limit TLS compilers and constrain the parallelism that they can uncover. Swarm addresses these issues, enabling SCC's novel techniques.

Renau et al.'s out-of-order spawn TLS [47] relaxed the requirement that task spawns must be serial, allowing speculative tasks to spawn children independently. This enables exploiting certain kinds of nested parallelism [34]. However, out-of-order spawn TLS is still restrictive, and does not achieve the benefits of selective aborts. Out-of-order spawn TLS also did not allow interleaving task timestamps in the manner SCC uses for progressive expansion. It also had a serial commit bottleneck, which it addressed by adaptively merging tasks if there are more tasks to run than cores. Thus, the system speculates at the coarsest granularity that fills the machine, building large task footprints that are prone to expensive aborts. By contrast, Swarm's

distributed queues manage many more speculative tasks than cores without merging tasks, enabling cheap selective aborts.

TLS compilers [5, 12, 13, 30, 38, 42, 56, 58, 63] are limited by the architectures they target. Like SCC, TLS systems as early as Multiscalar considered function calls and loop iterations as potential task boundaries [58]. However, without sufficient hardware support for fine-grain tasks, previous compilers focused on *selectively* parallelizing coarser tasks. POSH [34] also considered function calls and loop iterations, but focused on profiling to choose when *not* to use them as task boundaries, preferring to form sufficiently coarse tasks to amortize task overheads. The significant cost of unselective aborts also motivated techniques to reduce speculation. Du et al. developed models to statically estimate the likelihood of data dependences, to avoid spawning tasks that could frequently abort [13]. Zhai et al. developed compiler techniques to synchronize frequent data dependences instead of speculating on them [64]. By contrast, SCC speculates aggressively, leveraging Swarm's support for fine-grain tasks, selective aborts, and spatial hints to make speculation profitable even when there are frequent accesses to contentious data.

Most TLS compilers were evaluated with SPEC CPU2000, where they showed modest speedups. SCC achieves better speedups even on SPEC CPU2006 benchmarks, which are harder to parallelize [31, 38] (Chapter 4).

2.1.4 Speculative pipeline parallelism

Finally, prior work has also extended the DSWP-style loop pipelining techniques described above to parallelize loop iterations with unknown dependences. These systems include Speculative DSWP [57], DSWP+ [25], SMTX [43], and HTMX [15]. These systems pipeline a single loop across cores, speculating on unlikely data dependences and synchronizing on known ones. As we will shortly see, SCC is capable of achieving the benefits of these systems through fine-grain task decomposition together with spatial hints to isolate dependences. Moreover, SCC transforms the whole program and composes parallelism across loop nests and functions calls, whereas loop pipelining techniques can only parallelize a single loop level at a time.

Parallelized task structure	4 cores	36 cores	100 cores
TLS-style chain (Figure 2-1a)	4.0×	5.3×	4.9×
+ Decoupled spawn (Section 2.2.1)	3.5×	7.4×	7.7×
+ Nested fine-grain tasks (Section 2.2.2)	2.9×	8.2×	9.8×
+ Balanced spawner trees (Section 2.2.3)	3.6×	19.3×	31.3×

Table 2.1: Speedup of `mis` relative to original serial code.

```
1 for (int v = 0; v < numVertices; v++) {
2   if (state[v] == UNVISITED) {
3     state[v] = INCLUDED;
4     int* neighbors = getNeighbors(v);
5     int numNeighbors = getNumNeighbors(v);
6     for (int nbr = 0; nbr < numNeighbors; nbr++)
7       state[neighbors[nbr]] = EXCLUDED;
8   }
9 }
```

Listing 2.1: Sequential `mis` algorithm.

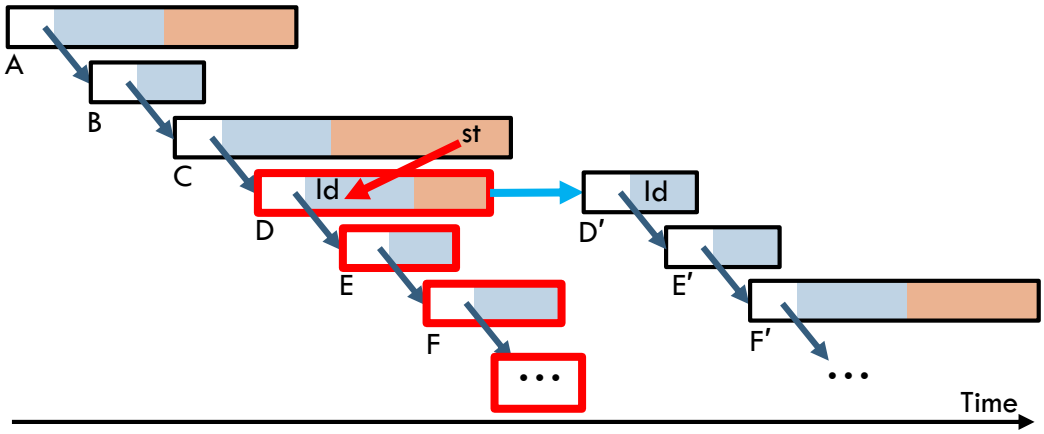
2.2 SCC motivation

We now illustrate the limitations of prior speculative parallelization approaches through a simple example, and progressively introduce key SCC features to overcome these limitations. Throughout these examples, we show that SCC's features require hardware support, and introduce Swarm, which provides sufficient support for these features.

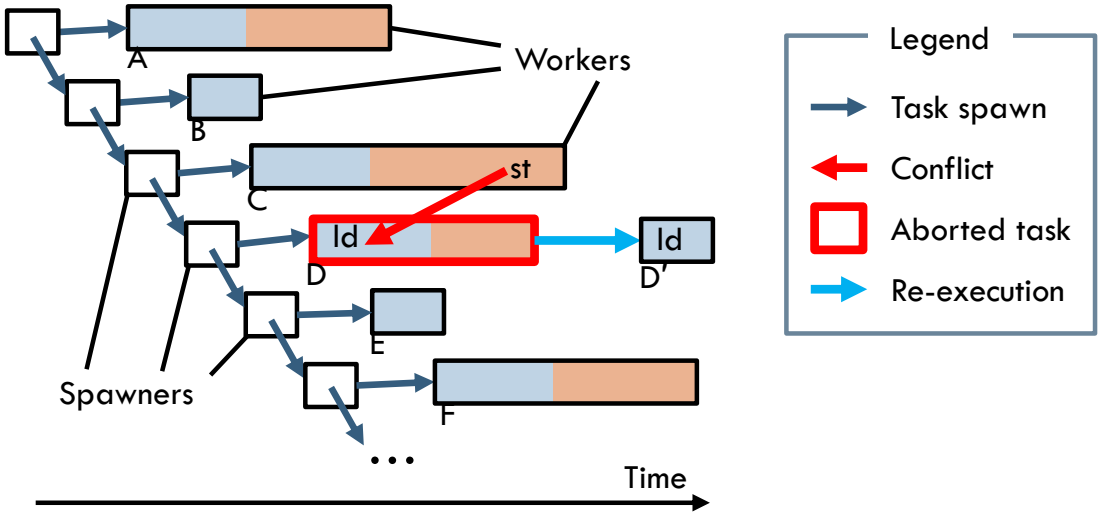
Consider the maximal independent set algorithm (`mis`). Given a graph, `mis` finds a set of nodes S such that no two nodes in S are adjacent, and each node not in S is adjacent to some node in S. Table 2.1 compares the performance of several parallelized `mis` variants on an R-MAT [11] graph with 1 M nodes and 95 M edges. Speedups are relative to a tuned serial baseline taken from PBBS [49]. (See Section 4.1 for methodology.) SCC extracts plentiful parallelism and keeps scaling to 100 cores by using *spawn decoupling* and *fine-grain tasks* to make aborts cheap, and *spawner trees* to keep many cores busy, as we will see next.

2.2.1 Decoupled spawn enables selective aborts

Listing 2.1 shows the `mis` algorithm. Each outer loop iteration, highlighted in blue, checks a vertex's state and includes it in the set if it hasn't already been excluded.



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



(b) With decoupled spawn, D aborts without aborting other tasks.

Figure 2-1: Execution timelines showing task structures for speculatively parallelizing `mis`, with each outer loop iteration constituting a task. A store in task C conflicts with a load of the same address in task D, resulting in aborts.

When a vertex is included in the set, the inner loop permanently excludes all the vertex's neighbors. The inner loop iterations, highlighted in orange, perform indirect stores, writing to addresses that are not known until runtime.

Since `mis`'s data dependences cannot be determined statically, we consider parallelizing `mis`'s outer loop using speculative parallelization. Prior TLS compilers would do so as depicted in Figure 2-1a: each task begins by spawning the next in the chain, and then runs the body of one outer loop iteration and, if executed, an entire inner loop. To preserve sequential semantics, the system tracks the set of memory addresses accessed by each task, and detects conflicting (i.e., out-of-program-order) accesses

among tasks. For example, in Figure 2-1a, task C writes to an address that was previously read by task D. This is an order violation or *conflict*. D must be *aborted* and re-executed so that it is correctly ordered after C. Any later-ordered task that received incorrect data from D must also abort. To accomplish this, all prior TLS systems conservatively abort *all later loop iterations* en masse [17, 21, 47, 50, 52]. These expensive mass aborts limit parallelism by delaying the ultimate execution of *all* later tasks. The TLS approach does not scale beyond 5.3×.

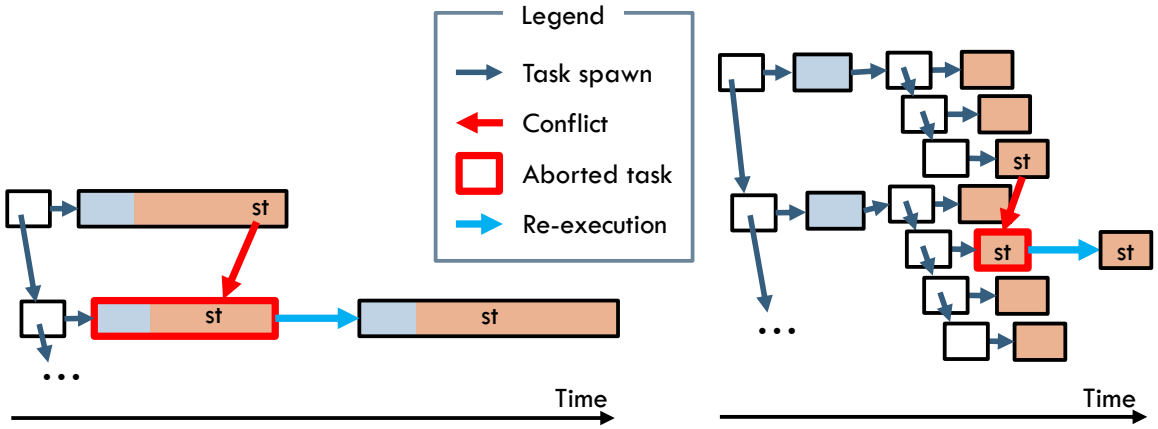
To address this problem, SCC leverages Swarm's mechanisms for *selective aborts*, wherein an aborting task would trigger additional aborts only for *dependent tasks* [27]. However, even with selective aborts, children are dependent on their parent, so they must abort if their parent aborts. For a task to abort independently of later tasks, it must avoid spawning children. To this end, SCC generates a tree of tasks rather than a simple chain. One of the simplest ways SCC can do this is as shown in Figure 2-1b: there is a chain of white *spawner* tasks, each of which spawns the next spawner in the chain, then spawns a *worker* task that executes the body of the loop. Now, workers such as D are decoupled from spawning, so D can abort without affecting later tasks.

A secondary benefit of decoupled spawn is that spawners can run on different cores than workers to exploit pipeline parallelism, similar to speculative DSWP [57]. Overall, decoupled spawn achieves a speedup of 7.7×.

2.2.2 Fine-grain tasks make aborts cheap

So far we have considered only parallelizing the outer loop in `mis`. But this leads to large tasks with long inner loops that are prone to conflicts and aborts. SCC can address this by parallelizing the inner loop too. Each inner loop iteration is tiny, so SCC divides the inner loop into tasks that each execute eight iterations. When a conflict arises, as shown in Figure 2-2b, it only requires aborting and re-executing a short task, a cheap operation. This improves the speedup to 9.8×.

In some situations, using fine-grain tasks to access contentious data has a secondary benefit: if each task accesses few addresses, SCC can map tasks to tiles in a locality-aware way, keeping contentious values in private caches rather than ping-ponging them across the chip (Section 3.5).



(a) Parallelize outer loop only **(b)** Parallelize both loops

Figure 2-2: Execution timelines depicting conflicting stores to the same address. Splitting `mis`'s inner loop iterations into parallel tasks enables resolving conflicts without aborting the entire outer loop iteration.

These fine-grain tasks require significant hardware support: The average task size is 89 cycles, so to keep 100 cores busy, the system must spawn, dispatch, and commit about one task per cycle. Prior hardware systems cannot achieve this throughput [27], so prior compilers use coarse tasks and limit aborts by selectively applying speculative parallelization in limited ways to loops or program segments where dependencies that would cause aborts are rare [8, 15, 34, 43, 61]. Instead, Swarm provides sufficient hardware to support fine-grain tasks with cheap aborts, enabling practical whole-program parallelization.

In addition, fine-grain tasks require new strategies in software: spawning a task by recording and sending a full thread context would add far too much overhead. SCC's efficient task closures drastically reduce each task's context (Section 3.6).

2.2.3 Balanced trees make fine-grain tasks scale

Spawning work quickly enough to keep many cores busy still requires new strategies for transforming application code. The serial chain of spawners in Figure 2-1b limits performance because only one core is spawning tasks at a time. SCC exposes greater parallelism through *loop expansion*, a transformation that creates balanced trees of spawners. For `mis`, SCC creates spawner trees for both inner and outer loop iterations, as depicted by the trees of white spawners in Figure 2-3. When a loop starts executing,

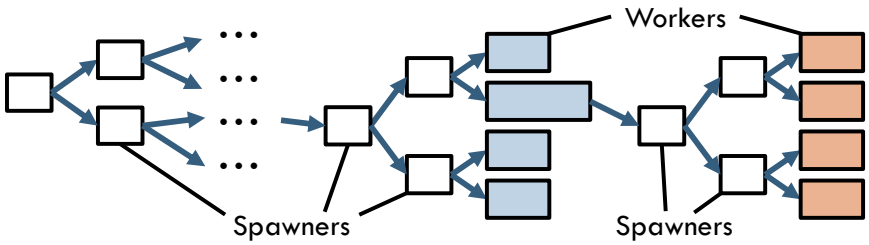


Figure 2-3: Balanced task trees improve parallelism.

```
void f() { g(0);  g(1);  g(42);  g(43); }
void g(int i) { for (int j = 0; j < 3; j++) h(i,j); }
```

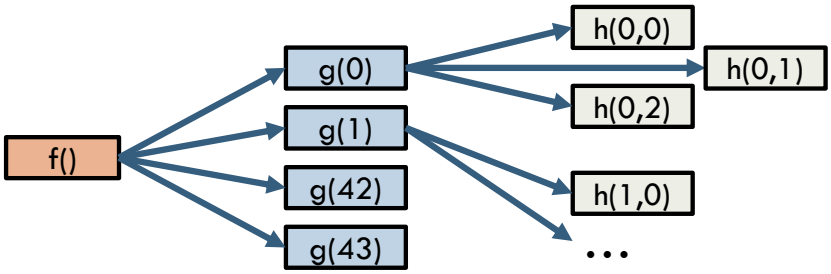


Figure 2-4: SCC exploits composition to parallelize across arbitrary calls and translation units.

spawner trees rapidly expand to distribute the load of both spawners and workers across the system. SCC thus runs `mis` 31.3× faster than the serial version.

Though `mis` loops are easy to transform into balanced trees, many loops have unknown bounds or exit conditions. SCC's *progressive loop expansion* achieves similar parallelism on these more complex loops (Section 3.4).

2.2.4 Nested parallelism enables composition

Unfortunately, programs do not spend all their time in long-running loops. Consider the code in Figure 2-4, which shows composition of function calls and loops as commonly arises in real-world programs. Function `f` calls function `g` 4 times, and each invocation of `g` runs function `h` 3 times. To exploit parallelism, SCC creates the task structure shown in Figure 2-4.[1] SCC compounds the parallelism within `f` and `g`, spawning $3 \times 4 = 12$ calls to `h` in parallel. This tree of tasks enables selective aborts of the `h` tasks. If `h` is large, it can be further divided into small tasks.

[1] Since SCC knows the loop in `g` is short, it does not use spawner tasks. `h` is directly spawned from worker tasks running `g` as shown in Figure 2-4.

Achieving this approach requires two key ingredients. First, it requires a way to compose parallelism *across translation units*: functions `f`, `g`, and `h` may be in different files, making it infeasible to have global view of control flow. Second, it requires a flexible, composable ordering mechanism that allows spawning all these tasks in parallel while preserving their order semantics.

This parallel nesting is not supported by most prior systems for speculative parallelization. The closest prior work is Renau et al.'s TLS with out-of-order spawn [47]. This prior system could exploit some forms of nested parallelism, but without the hardware support needed for low-overhead fine-grain tasks, it instead merged tasks into coarser-grain units, missing the opportunity to exploit fine-grain parallelism. Furthermore, it places restrictions on spawn order that would make SCC transformations such as progressive expansion impossible. By contrast, Swarm provides sufficient support for SCC to implement this nested parallelism strategy.

2.3 Swarm architecture

SCC targets the Swarm architecture [27, 28]. Specifically, it uses the Fractal version of Swarm [55], which supports nested parallelism and incorporates spatial hints [26] for locality-aware execution. We first explain Swarm's execution model, then highlight the key microarchitectural features SCC must exploit to achieve good performance.

2.3.1 Swarm execution model

A Swarm program consists of a set of timestamped tasks. Any task can spawn children tasks, tagging them with a timestamp greater than or equal to its own. Swarm guarantees that tasks appear to run in timestamp order; same-timestamp tasks run atomically (i.e., they do not interleave). Swarm provides precise exceptions [29] so tasks can run unrestricted code (e.g., system calls).

Fractal extends Swarm by placing tasks in a hierarchy of nested *domains*. Within a domain, tasks are ordered with timestamps as before. Additionally, each task may create a single subdomain and spawn children into that subdomain. Fractal guarantees atomicity among domains: all tasks in a domain along with its creator appear as a single atomic unit.

Domains make composition easy, as they allow separately parallelized code regions to use separate domains with independent timestamp schemes. Tasks from multiple domains run in parallel. For example, in `mis`, each outer loop iteration can create a subdomain in which the inner loop will run. This way, each outer loop iteration appears as an atomic unit ordered by its timestamp in the outer domain.

2.3.2 Swarm microarchitecture

Swarm introduces modest changes to a tiled multicore, shown in Figure 2-5. Each tile has a group of cores. The cores in a tile share an L2 cache, and each tile has a slice of a fully shared L3 cache. Each tile is augmented with a *task unit* that queues, dispatches, and commits tasks.

To scale, Swarm uses *fully distributed mechanisms*. Local task units queue and dispatch tasks to cores autonomously. When a core spawns a task, it communicates only with its local task unit, which buffers newly created tasks. Tiles balance load by sending tasks to each other using point-to-point communication, without any centralized scheduler. Thus, cores can spawn many tasks in parallel, and do not need to stall on memory accesses or for cross-chip communication latency when spawning tasks.

To prevent order from limiting throughput, Swarm speculates thousands of tasks ahead of the earliest active task. To this end, Swarm has large hardware queues, e.g., it can buffer up to 256 tasks waiting to execute per tile (64 per core) and up to 64 tasks waiting to commit per tile (16 per core). Swarm uses these queues to achieve high commit throughputs efficiently: tiles periodically communicate to find the earliest unfinished task in the system, then commit all earlier tasks. This commit protocol amortizes communication among many tasks. It is the only global operation in the system, but it is a reduction, so overheads scale logarithmically with the number of tiles and are negligible at this scale [27].

Swarm uses eager (undo-log-based) version management and eager (coherence-protocol-based) conflict detection with Bloom filters, similar to LogTM-SE [62]. Swarm extends these techniques to implement selective aborts. Finally, Swarm leverages *spatial hints* [26] to perform locality-aware execution. A hint is an integer optionally given when a task is spawned. Software should give the same hint to tasks likely to

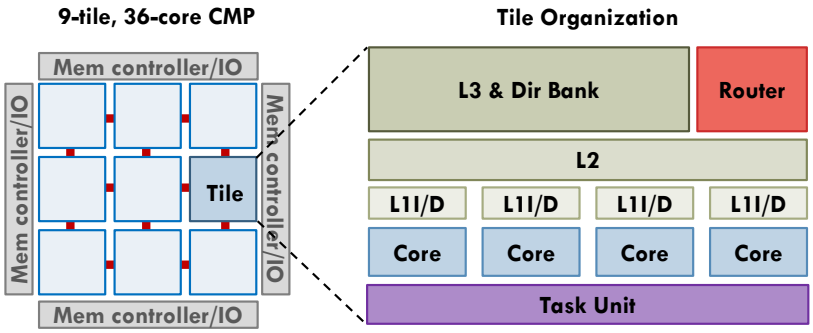


Figure 2-5: 36-core/9-tile Swarm processor configuration.

access the same mutable data (Section 3.5). Swarm runs same-hint tasks in the same tile to exploit locality.

These techniques allow Swarm to execute work aggressively out-of-order, with speculation windows of thousands of tasks, i.e., hundreds of thousands of instructions. This makes it advantageous for programs to spawn work early. We thus design SCC to spawn work aggressively, and carefully manage live data values to ensure cores do not need to access shared memory frequently in order to spawn or run tasks.

Chapter 3

The Swarm C/C++ Compiler

3.1 SCC overview

Unlike prior work [8, 15, 34, 43, 61], SCC does not use profile-guided heuristics to limit parallelization. SCC *parallelizes the entire program*, exposing fine-grain parallelism starting from the first instruction of `main`. To do this, SCC first splits code into tasks at fine granularity and tags tasks with timestamps to record program order. After this, SCC is free to aggressively transform the code to spawn tasks in *any* order to improve parallelism, as Swarm hardware will guarantee apparent program-ordered execution.

Figure 3-1 gives an overview of SCC's main components. SCC adds transformation passes to the LLVM/Clang compiler toolchain. All of SCC's passes are *intraprocedural*, that is, a compiler performs them on one function at a time, without relying on expensive *inter*procedural analyses. Thus, SCC compilation times stay small and proportional to code size. SCC's passes run towards the end of the LLVM middle-end, after target-independent optimizations, so SCC does not impede the performance of standard compiler optimizations. SCC parallelizes this sequential IR in four phases:

1. Reducing memory dependences in sequential code: To improve parallelism and enable efficient function spawns, SCC first optimizes the allocations of local variables and avoids using a shared call stack (Section 3.2). These optimizations include privatizing local variables scoped within loops, and transforming all function calls whose continuations use the function's return value into *continuation-passing*

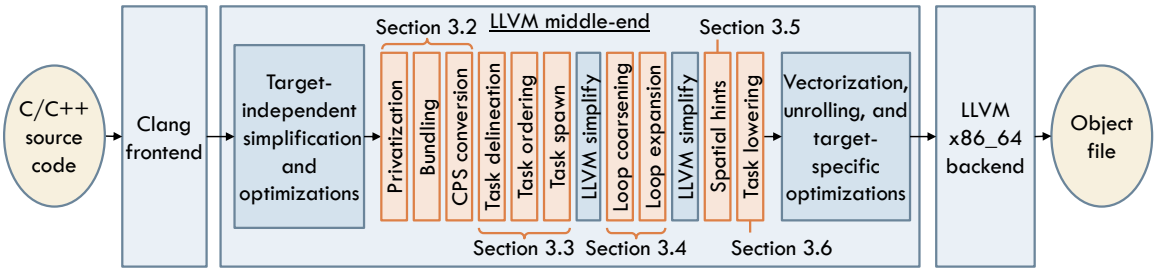


Figure 3-1: Structure of SCC. SCC is implemented by adding a series of passes to LLVM's middle-end. Transformations newly implemented for SCC are highlighted in orange.

style (CPS): functions are passed a continuation that encapsulates the computation that should run when the return value is known [2]. CPS eliminates the need to store stack frames and return addresses. Instead, data needed by the caller's continuation is placed in a register- or heap-allocated *closure*.

These transformations are safe for general, sequential code. In the common case, they reduce the state associated with each task (to a few register values rather than a full thread context) and avoid false-sharing conflicts that would arise from a shared stack.

2. Fine-grain task delineation: SCC spawns potentially parallelizable tasks at the finest granularity, including converting all loops and loop nests into tasks, with one task per iteration (Section 3.3.1). SCC tags these tasks with timestamps to reflect their original program order (Section 3.3.2).

3. Task refinement: SCC spawns tasks out of order to improve parallelism (Section 3.3.3). It transforms loop control and body tasks to improve performance (Section 3.4). SCC also adds spatial hints to exploit locality (Section 3.5).

4. Task lowering: Finally, SCC *lowers* (i.e., translates) the timestamp-tagged tasks to ordinary LLVM IR. This requires producing a separate function and closure for each task. SCC optimizes closures to reduce overheads (Section 3.6). After this, the LLVM backend generates executable code.

Prior to lowering, SCC extends LLVM's intermediate representation (IR) to represent parallel tasks within a function's control-flow graph (CFG). This representation is similar to Tapir [48], which was used to optimize conventional multithreaded code. By incorporating parallelism in the IR, SCC can perform parallelizing transformations as

simple, modular compiler passes, and leverage the sophisticated analysis machinery of LLVM to perform optimizations *across tasks* (e.g., induction-variable elimination in a parallelized loop (Section 3.4)). This approach provides high-quality code generation.

As shown in Figure 3-1, SCC runs conventional LLVM optimizations between these passes to simplify and optimize the IR. For example, they remove dead code that may have arisen because of loop transformations.

3.2 Eliminating the shared call stack

Before dividing code into parallel tasks, SCC first performs several transformations that reduce conflicts and enable reduced task overheads. Specifically, SCC avoids the use of a globally shared function call stack, which would become a point of contention for fine-grain parallel tasks. When tasks run, they have thread-private stacks they can use for register spills, but these thread-private stacks are not used for values shared among tasks or to implement function calls. To accomplish this, we need to perform two transformations to the code: *bundling* variables to the heap if they must be shared among tasks through memory, and transforming the code into continuation-passing style.

3.2.1 Variable bundling

Bundling variables to the heap is rare: it is not used for local values that could be promoted to registers, or spilled to thread-private stacks when registers are full. It happens only for variables the program takes pointers or references to. SCC bundles all such variables together within a single function and allocates them as a single heap chunk.

Cactus stacks as used in fork-join parallel languages like Cilk [16] are the closest technique to SCC's bundling. However, bundling gives SCC two important benefits over cactus stacks. First, it is selective: whereas each Cilk task takes a (heap-allocated) cactus stack frame, in SCC most functions do not incur any heap-allocation overheads. Second, by placing shared mutable data in the heap, bundling separates this mutable data from private local values that will be allocated on thread-private stacks. This avoids spurious conflicts due to our cache-line-based conflict detection.

3.2.2 Continuation-passing style conversion

Continuation-passing style (CPS) conversion avoids using the stack as a record of function call frames and eliminates the notion of a function returning to its caller. This means there's no need to allocate memory on each function call to save stack frame pointers or return addresses. Instead, it enables SCC to use a new task and function calling convention which allows us to spawn some tasks and functions with zero memory accesses, passing values purely through registers (Section 3.6). This optimization is crucial to keep overheads low with SCC's very fine-grain tasks.

CPS conversion is a general and well-studied transformation in programming languages [40, 54], and is used in compilers for functional languages [2, 3, 51]. Although some explicitly parallel programming models have used CPS conversion [19, 32], we are the first to use this transformation in a system for speculative parallelization.

CPS conversion modifies each function so that it can accept a continuation, and also modifies every callsite to pass a continuation if needed. Specifically, each function takes an extra argument, which is a pointer to a continuation's *closure*. The closure is an object allocated on the heap. Its first field is a function pointer to the continuation, and subsequent fields are values captured by the caller that the continuation uses when it runs. At the end of its execution, instead of returning, the function calls the continuation and passes the return value of the function, if it had one.

3.2.3 Privatization

When bundling variables to the heap, SCC detects variables scoped within the body of a loop and *privatizes* them so that a separate space is allocated for each iteration of the loop. This avoids false sharing conflicts between loop iterations. We find that moving the declarations of variables only used within a loop body to enable this transformation is easy (Section 4.1.2), and it is also a good practice in modern programming.

After the transformations of privatization, bundling, and CPS conversion, the sequential code is ready for parallelization. Many sources of false dependences have been removed and functions calls no longer use a shared call stack, making it easier to spawn many tasks in parallel without memory allocation in the common case.

3.3 Whole-program fine-grain parallelization

Now that the code has been transformed to be more amenable to parallelization, SCC must record the program order specified by the sequential code among tasks within functions and loops. It uses Fractal to compose parallelism across function calls and loop nests while preserving sequential semantics. This then allows SCC spawn tasks aggressively and out of order to expose parallelism. We now explain the code transformations SCC performs one function at a time to accomplish composable fine-grain parallelization across an entire program.

3.3.1 Task delineation

SCC divides code into tasks by turning each loop iteration, each function call, and each function call's continuation into a separate task. Like prior work, we find that this approach naturally limits tasks sizes to be fairly small [58]. These heuristics suffice for compiling most code. However, some tasks remain that are coarser than needed. To exploit more parallelism, we offer the programmer annotations to split tasks more finely (Section 3.7). SCC further transforms loop tasks to improve parallelism, as Section 3.4 will discuss.

3.3.2 Task ordering

SCC orders tasks with a combination of timestamps and Fractal domains. This process starts with the *control flow graph* (CFG) of a single function, a directed graph whose nodes are basic blocks. If the function has any internal tasks, SCC creates a domain for the function's tasks, and orders the tasks within each domain using timestamps. To do this, first, SCC collapses strongly connected components (due to, for example, loop backedges) into a single node, so that the remaining CFG becomes acyclic.

SCC then topologically sorts the nodes of the acyclic CFG, and assigns timestamps to tasks in topological order, treating each collapsed node as a single task. Thus, it is guaranteed that these timestamps reflect program order.

SCC then creates a Fractal subdomain for each collapsed node in the acyclic CFG. SCC gives all tasks within that node timestamps that reflect program order. To do

this, SCC may recursively repeat the timestamp assignment algorithm, topologically sorting sub-CFGs within each collapsed node.

3.3.3 Task spawn decoupling

After dividing the sequential code into tasks, SCC determines, for each function call or loop task T, whether T spawns next following task U, or whether T and U should be spawned in parallel by a common parent task. The latter option will decouple the tasks so they can be aborted selectively (Section 2.2.1). Therefore, SCC follows a heuristic that favors decoupling: T spawns U as a descendant only if task T produces *live-out register values* that need to be directly passed to later task U. Otherwise, tasks T and U are spawned as decoupled sibling tasks.

For example, the continuation of a function callsite is only passed to the continuation to be spawned later if the continuation needs the callee's return value. Otherwise, the caller spawns the function call and continuation in parallel. Similarly, if a loop continuation depends directly on a local value produced by a loop, the last iteration of the loop spawns the continuation. Otherwise, loops are spawned in parallel with their continuations.

3.4 Efficient loop expansion

SCC adopts a multifaceted strategy to parallelize loops. Central to this strategy is the use of *spawners* as described in Section 2.2: small tasks that do little work and are thus unlikely to abort, which are responsible for spawning *workers* that execute the bodies of loops. SCC introduces three compiler transformation strategies to *expand* loops to generate spawners: balanced-tree expansion, progressive expansion, and chain expansion. Progressive expansion is the most critical for fine-grain parallelization of programs with irregular control flow. We explain these strategies next. SCC also coarsens loop tasks to reduce overheads, which we discuss briefly at the end of this section.

3.4.1 Balanced tree expansion

In cases like `mis` (Section 2.2.3), where the number of iterations is known at the start of the loop, building balanced spawner trees is easy: each spawner is responsible for a consecutive range of iterations, which it divides evenly across children spawners. When a spawner has a small-enough range (e.g., 8 worker tasks in our implementation), it spawns worker tasks for each iteration. Spawner trees grow exponentially and quickly fill the machine with work, as shown in Figure 2-3. To avoid choking the machine with tasks, each spawner is timestamped with the start of its range, which prioritizes workers and spawners properly and expands the loop gracefully.

3.4.2 Progressive loop expansion

Progressive expansion generates exponentially growing trees of spawners *even on loops with irregular control flow*, without knowing how many iterations to run. It does so by creating a tree of *speculative spawner* tasks.

Figure 3-2 shows progressive expansion in action on a loop with an unknown termination condition. Spawner tasks are shown in gray and loop iteration tasks are shown in blue. The number within each spawner task denotes its timestamp. Each spawner task spawns both multiple loop iteration tasks and multiple spawner tasks for later iterations.

To handle unknown termination conditions, progressive expansion transforms each loop iteration to set a new `done` flag indicating that the loop is finished if it discovers that it should be the last iteration. This is shown in the lower left inset of Figure 3-2. Loop iteration and spawner tasks start by reading this flag and exit without doing anything if the loop has already finished. Thus, the growing tree of spawner tasks terminates shortly after any iteration discovers the loop is finished.

For simplicity, Figure 3-2 shows spawners that each spawn two loop iterations directly, then spawn two child spawners with doubling stride. SCC initiates loop execution by spawning the initial spawner task, `spawner(0, 2)`, which will lead to the eventual spawning of all loop iterations. In our evaluation, each spawner spawns four loop iterations and four spawners.

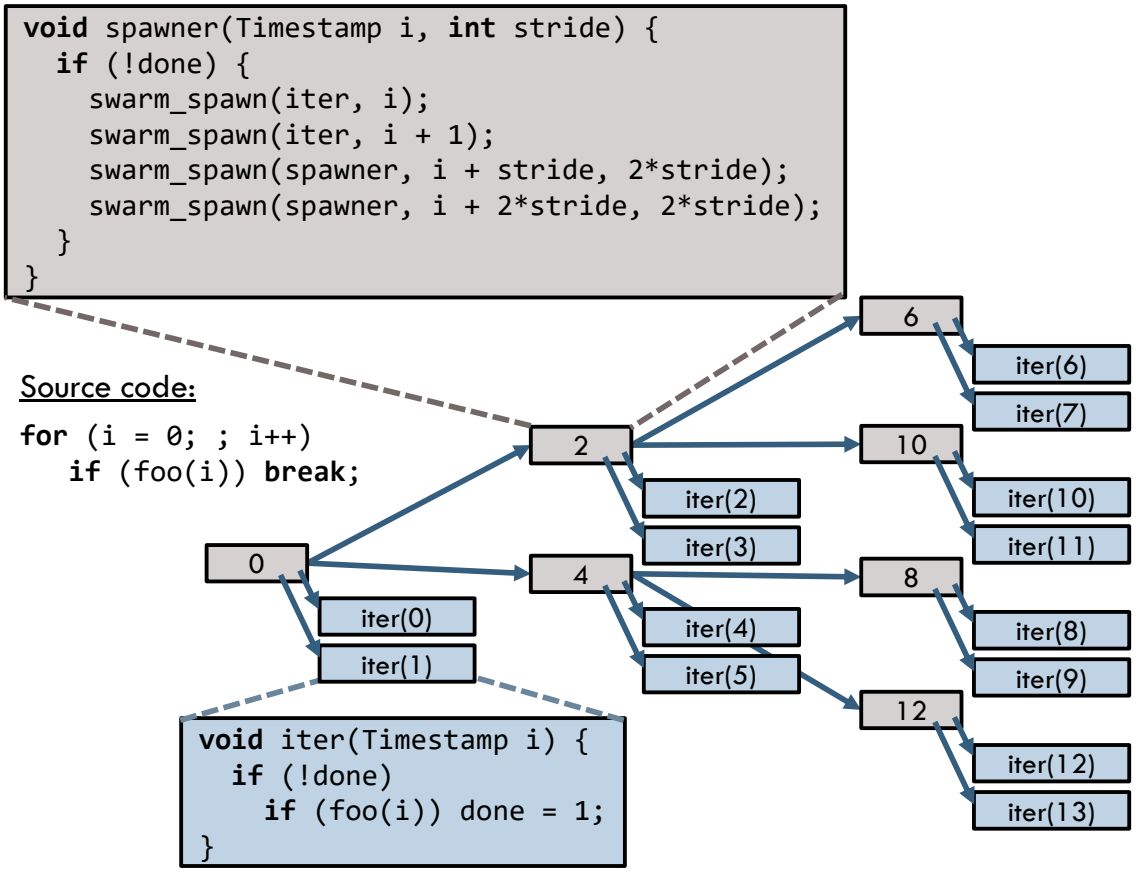


Figure 3-2: The progressive expansion task structure for parallelizing loops.

Progressive expansion is performing a form of *control speculation*, optimistically creating more spawners and loop iteration tasks than may be needed. By generating the `done` flag and treating it as an ordinary memory-resident variable, progressive expansion reuses Swarm's existing mechanisms for scalable conflict detection. 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 conflict checks and abort any tasks that misspeculatively ran past the end of the loop.

Progressive expansion relies on Swarm's ability to spawn tasks out of program order, using explicit timestamps. For example, in Figure 3-2, the spawner children of spawners 2 and 4 are *interleaved* (with timestamps 6 and 10, and 8 and 12). This is not possible in TLS systems, where spawned threads must be *immediate successors* of their parent. In general, to achieve a logarithmic critical-path length, at each level of the spawner tree, the number of iterations in each spawner's subtree must decrease by a constant factor. In order to do this without knowing how many iterations there will

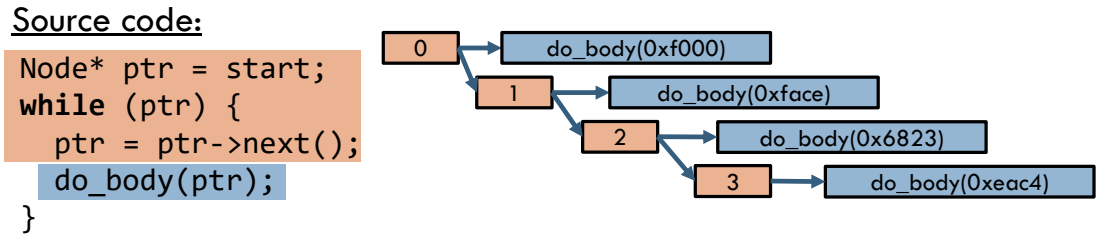


Figure 3-3: The general chained loop task structure.

ultimately be, child spawners must be given interleaved iterations.

By generating spawner trees, progressive expansion and balanced tree expansion stand in contrast to prior TLS compilers, which spawn iterations of any loop serially.

3.4.3 Chain expansion

SCC's spawner trees depend on the ability to eliminate loop-carried dependences, i.e., a value produced in each iteration and strictly needed in the next. We find the vast majority of hot inner loops lack such dependences or can be rewritten to remove them (e.g., by using induction-variable elimination [1, Sec. 9.1.8] [1]). For the remaining loops, SCC performs *chain expansion*, forming spawner chains that exploit pipeline parallelism across iterations as in prior TLS compilers.

Figure 3-3 shows chain expansion in action on a loop that traverses a linked list, with a loop-carried dependence on `ptr`. Chain expansion divides each loop iteration into two parts: the slice that computes the loop-carried dependence, shown in orange in Figure 3-3, and the slice that either consumes or is independent from the loop-carried dependence, shown in blue. SCC then produces a tight chain of orange tasks that compute the loop-carried dependence quickly to minimize the critical path. Each of these tasks also spawns a blue task. Blue tasks are off the critical path and overlap, obtaining some parallelism. Each blue task can be long, and often contains function calls or inner loops that SCC breaks into tasks, exploiting nested parallelism across iterations.

[1] We find that recent versions of LLVM have sufficiently advanced infrastructure for identifying and rewriting induction variables that, unlike prior work [34], we did not find a need to perform software value prediction to eliminate dependencies on induction variables.

3.4.4 Loop task coarsening

To reduce overheads, SCC identifies inner loops that do little work per iteration, and *coarsens* the tasks associated with the loop so each task executes several loop iterations. When coarsening, SCC uses LLVM's loop analyses to detect strided accesses to place iterations that share a cache line in the same task (e.g., with 64-byte cache lines and accesses that stride 48 bytes per iteration, SCC coarsens by a factor of 4 and generates prolog and epilog code so that each task covers its own 3 cache lines). This avoids false sharing across tasks, which could cause spurious conflicts.

3.5 Locality-aware speculation

Because SCC breaks programs into fine-grain tasks, some tasks access only one or a few locations in memory. This allows implementing simple heuristics to generate spatial hints, so that Swarm can ensure tasks that touch the same contentious data all run on the same chip tile.

SCC generates spatial hints in the following steps. First, it analyzes stores to identify potentially contentious variables, e.g., variables that appear to be modified repeatedly across the iterations of a loop, or which have been annotated as such by the programmer (Section 3.7). Then, SCC identifies tasks that write to a single contentious memory location as candidates to be given spatial hints. Finally, SCC checks if it can safely hoist the address computation for the memory access into the parent task. If the address computation is successfully hoisted, then SCC uses the *cache line address* (computed by right-shifting the address) as the hint.

SCC uses simple heuristics to filter what variables it may consider contentious. It ignores termination variables, e.g., as generated by progressive expansion (Section 3.4.2). If LLVM's analyses can determine that all potentially contentious stores in a task write to fields within a single memory object, SCC uses the object's base pointer.

3.6 Task lowering

Task lowering constructs an explicit *closure* for each task. If a task's live-in values 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 when the parent spawns the task, avoiding any accesses to shared memory. Because SCC produces fine-grain tasks, this is usually the case for leaf tasks, which make up the majority of tasks in SCC programs. If task 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 from memory.

SCC performs two optimizations to reduce closure sizes:

1. *Live-in sinking* finds task live-ins that can be cheaply recomputed from other available values (e.g., multiple addresses computed by adding constant offsets to the same pointer) and sinks the computation into the task.
2. *Loop closure sharing* checks whether tasks in a loop would need to allocate closures on the heap. If so, it allocates a partial closure that holds all loop-invariant live-ins. All loop iterations read from this single location, achieving good L1 hit rates and greatly reducing allocation overheads.

Together with SCC's techniques to avoid using memory to communicate arguments or return values when spawning functions (Section 3.2), these optimizations make heap-allocated closures rare in most programs.

3.7 Program annotations

SCC introduces simple code annotations that can be used to encourage the compiler to split tasks into finer granularity. None of these annotations affect program semantics. We add these annotations sparingly, after profiling to find the code that access the most frequently updated mutable data structures, which will create contention. Future versions of SCC could use profiling to offer suggestions for where to place task annotations [59], or add fine-grain task boundaries automatically based on identifying contentious data.

A common pattern is for a stretch of straight-line code, e.g., within the body of a loop, to first read uncontentious, infrequently changing data, then perform some

computation, and finally perform an update of some mutable data. If the mutable data is frequently updated, this contention can cause wasteful aborts if the entire block of straight-line code aborts, forcing the uncontentious reads and computation to re-execute. To make aborts less frequent and cheaper, the programmer may suggest additional task boundaries to make tasks more fine-grain. This allows the latter part of the code that accesses contentious data to potentially better exploit spatial locality through spatial hints (Section 3.5), and to abort cheaply, without aborting a larger code block.

These annotations improve performance, but *do not change program semantics*: the program retains its sequential, deterministic behavior regardless of where annotations are added.

3.8 Putting it all together

Listing 3.1 shows one of the two hottest loops in `astar`, a SPEC CPU2006 benchmark that finds paths in a 2D grid. This loop illustrates how progressive expansion, hint generation, and program annotations work together.

Because the loop's code is highly repetitive, Listing 3.1 omits duplicated code blocks (highlighted in green). Each loop iteration visits a node identified by `index`. For each of the node's eight neighbors in the 2D grid, identified by `index1`, the code checks whether the neighbor should be added to a queue `bound2p` to be visited later. On average, about one of the eight neighbors is enqueued to `bound2p` per iteration.

We insert task boundary annotations surrounding each of the eight repeated neighbor-checking blocks, exploiting parallelism among these blocks. Most neighbor-checking tasks will not decide to enqueue the neighbor, so they are read-only and parallelizable.

However, to avoid frequent, expensive aborts, we must deal with contentious access to the queue. We further split the enqueue in line 7 into its own task. SCC executes tasks as shown in Figure 3-4: each iteration starts with a single task, in blue, which spawns the eight neighbor-checking tasks, in green. When a neighbor must be enqueued, a child task is spawned to perform the neighbor enqueue (line 7), in orange.

```
1  int bound2l = 0;
2  for (i = 0; i < bound1l; i++) {
3    index = bound1p[i];
4    index1 = index-yoffset-1; // NW neighbor
     // 1st of 8 identical code blocks:
5    if (waymap[index1].fillnum != fillnum
6        && maparp[index1] == 0) {
7      bound2p[bound2l++] = n;
8      waymap[index1].fillnum = fillnum;
9      waymap[index1].num = step;
10     if (index1 == endindex) {
11       flend = true;
12       return bound2l;
13     }
14   } // End of 1st identical code block.
15   index1 = index-yoffset;   // N neighbor
     // 2nd of 8 identical code blocks appears here.
26   index1 = index-yoffset+1; // NE neighbor
     // 3rd of 8 identical code blocks appears here.
37   index1 = index-1;         // W neighbor
     // 4th of 8 identical code blocks appears here.
48   index1 = index+1;         // E neighbor
     // 5th of 8 identical code blocks appears here.
59   index1 = index+yoffset-1; // SW neighbor
     // 6th of 8 identical code blocks appears here.
70   index1 = index+yoffset;   // S neighbor
     // 7th of 8 identical code blocks appears here.
81   index1 = index+yoffset+1; // SE neighbor
     // 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

Task splitting lets SCC generate effective spatial hints (Section 3.5). All neighbor-enqueue tasks use a spatial hint based on the address of `bound2l`, localizing them to execute on a single tile. Swarm will abort and re-execute these tasks as necessary to preserve sequential semantics, as shown in Figure 3-4. Splitting off neighbor enqueues and delegating them to one location keeps accesses to the queue local and cheap.

Finally, to keep many cores busy, SCC must parallelize across loop iterations. Progressive expansion is needed to allow for spawning iterations of this loop in advance: reaching a particular end-of-map cell identified by `endindex` can cause the

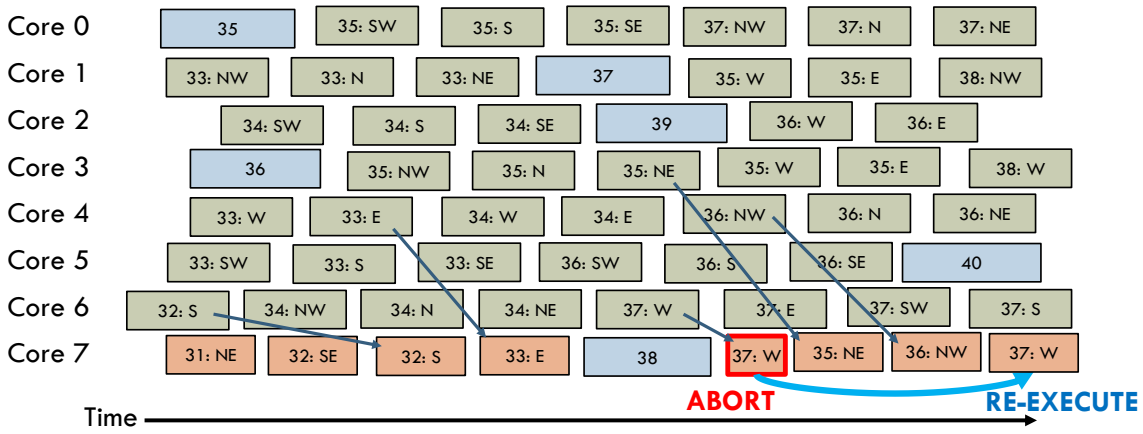


Figure 3-4: Execution timeline of `astar` with task colors matching Listing 3.1. Numbers indicate loop iterations, and tasks associated with one of the eight neighbors are indicated with the cardinal direction of the neighbor. Arrows indicate spawns of neighbor-enqueue tasks.

loop to terminate on any iteration. However, reaching `endindex` happens very rarely, and it is profitable to speculatively spawn many loop iterations in parallel.

Chapter 4

Evaluation

4.1 Experimental methodology

4.1.1 Simulated hardware

We use a cycle-level simulator based on Pin [35, 39] to model Swarm systems with parameters in Table 4.1. Unless otherwise noted, experiments use in-order cores. Swarm parameters match those of prior work [26, 27, 28]. 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 keep per-core cache sizes and queue capacities constant across system sizes.

4.1.2 Benchmarks

We use SPEC CPU2006 C/C++ benchmarks to evaluate SCC. These benchmarks comprise real-world applications [24] that are hard to auto-parallelize: initial attempts to apply TLS showed upper-bound speedups below 1% [31]. Packirisamy et al. applied state-of-the-art TLS techniques to SPEC CPU2006 programs, showing moderate speedups (1.6× at 4 cores, 1.91× at 8 cores) [38]. We use seven of the 19 benchmarks, listed in Table 4.2. From the remaining benchmarks, two (`gcc`, `sjeng`) are not yet correctly compiled by SCC, and the remainder use features that SCC cannot yet parallelize, so SCC falls back on executing them in long, serial tasks, obtaining

Cores	36 cores in 9 tiles (4 cores/tile), 2 GHz, x86-64 ISA; single-issue in-order scoreboarded (stall-on-use) [26]; or Haswell-like 4-wide OoO superscalar [20]
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 8-way, inclusive, 7-cycle latency
L3 cache	9 MB, shared, static NUCA [33] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	3×3 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [60])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (2304 total), 16 commit queue entries/core (576 total)
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [10] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Fractal time	128-bit virtual times, tiles send updates to virtual time arbiter every 200 cycles
Spills	Spill 15 tasks when task queue is 85% full

Table 4.1: Configuration of the 36-core system.

Benchmark	Lines of code	Modified lines	1-core cycles per task
`429.mcf`	1574	None	635
`433.milc`	9575	+17, -10	372
`456.hmmer`	20680	+13, -9	266
`462.libquantum`	2605	None	610
`470.lbm`	904	+1, -1	2649
`473.astar`	4285	+32, -141	31
`482.sphinx3`	13128	+3, -2	146

Table 4.2: SPEC CPU2006 benchmarks used in the evaluation. Lines of code exclude comments and whitespace.

no speedup. These include C++ exception-handling library code (used by C++ benchmarks), function pointers stored in application data structures (`gobmk` and `h264ref`), and `sigsetjmp`/`siglongjmp` (`perlbench`). We expect to add support for these features in future work.

We compare the SCC-compiled version of each benchmark with the original serial version, compiled with `-O3` on unmodified LLVM/Clang 5.0 (SCC is based on

LLVM/Clang 5.0). We have verified that SCC-compiled benchmarks deterministically produce the same result as the serial version.

We evaluate all benchmarks with the `ref` (largest) inputs. Since SPEC CPU2006 benchmarks run for a very long time, we evaluate a sample period of their execution corresponding to 2 billion dynamic instructions of the serial version after fast-forwarding 10 billion instructions to skip initialization. This sample interval suffices to cover a long, representative period of steady-state execution. None of the benchmarks spend the sample period on a single hot loop: all enter and exit from loops and execute different code many times, allowing us to evaluate SCC's performance across program phases.

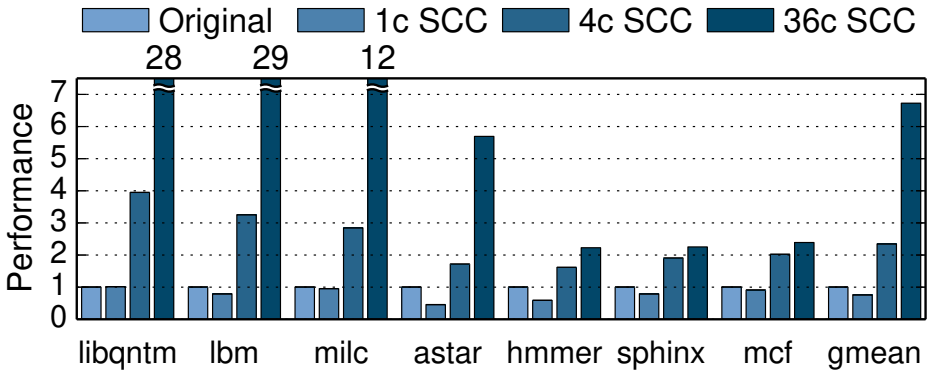
To ensure we compare the same region of execution regardless of compiler transformations, we instrument the entry point of certain functions with *heartbeats*, and count the heartbeats in our simulator. We find the heartbeat counts closest to 10 and 12 billion instructions in the original serial version, and simulate that interval.

We modify the source code of some of the benchmarks to help the compiler uncover more parallelism. Table 4.2 reports the lines changed, a small fraction of the program's lines of code in all cases. In `hmmer`, `astar`, and `sphinx3`, these modifications are SCC annotations to break up tasks as explained in Section 3.7. We also manually perform loop fission in `hmmer` to allow SCC to better divide different striding memory accesses into parallel tasks that access separate cache lines (Section 3.4.4). We'll analyze the impact of these manual task-splitting code modifications later. `lbm` and `milc` use simple modifications to avoid false sharing. In `milc`, we move the declarations of variables into loops, scoping the variables more tightly, when they are used to hold short-lived values within each loop iteration. This enables SCC's privatization to avoid false sharing conflicts (Section 3.2.3).

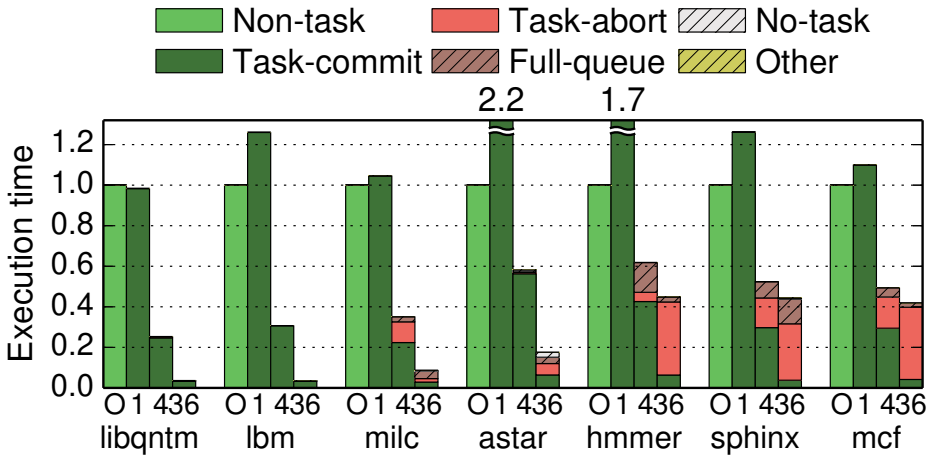
4.2 Results

4.2.1 SCC performance

Figure 4-1a reports the performance of SCC normalized to original serial code running on a 1-core system (higher is better). On 4 cores, SCC scales all benchmarks well,



(a) Performance vs. original serial code



(b) Normalized execution time breakdown

Figure 4-1: Comparison of (a) performance and (b) execution time of SCC on 1-, 4-, and 36-core systems, normalized to the Original serial code on a 1-core system.

achieving gmean 2.4× speedup over the original serial code. SCC achieves linear speedups to 36 cores on `libquantum` and `lbm`, 28× and 29×, respectively. SCC achieves gmean speedup of 6.7× at 36 cores compared to the original serial code.

Figure 4-1b provides further insight into these results. Each bar shows the execution time (lower is better) of SCC at 1, 4, and 36 cores, relative to the execution time of the original serial version. The 1-core SCC bars show that *SCC introduces modest overheads* of up to 26% for five of the benchmarks. Overall, 1-core SCC is gmean 31% slower than the serial versions due to higher overheads in `astar` and `hmmer`. These overheads are reasonable when considering that SCC divides these programs into very fine-grain tasks, tens to few hundred cycles long, as shown in Table 4.2. In return, short tasks make aborts cheap and confer large speedups: SCC is 2.8× faster at 4 cores, and 8.8× faster at 36 cores, than it is at 1 core.

Figure 4-1b also shows a breakdown of how cycles are spent. In SCC, cores execute *(i)* tasks that later commit or *(ii)* later abort, and spend cycles *(iii)* stalled on a full commit queue, *(iv)* idle because there are no tasks available to run, or *(v)* in other overheads.

Figure 4-1b shows that, on all benchmarks at 4 cores, cores spent most of their time executing useful work that will commit. Because `hmmer`, `mcf`, and `sphinx3` do not scale beyond 4 cores, their execution time changes little from 4 to 36 cores, with the additional cores mainly running more speculative tasks that abort. The other four benchmarks scale to 36 cores, with aborts making up a minority of execution time. Furthermore, little time is spent idle. This demonstrates that, when a benchmark has sufficient parallelism, SCC spawns useful tasks sufficiently ahead of time so that task scheduling is done off the critical path by task units, keeping cores busy actually executing tasks rather than waiting on tasks to arrive.

4.2.2 Benchmark analysis

Several benchmarks reveal interesting behaviors and show the need for speculative parallelization.

`lbm` and `libquantum` have plentiful parallelism in their inner loops, but compilers cannot statically parallelize all of them, as prior work notes [38]. SCC scales these

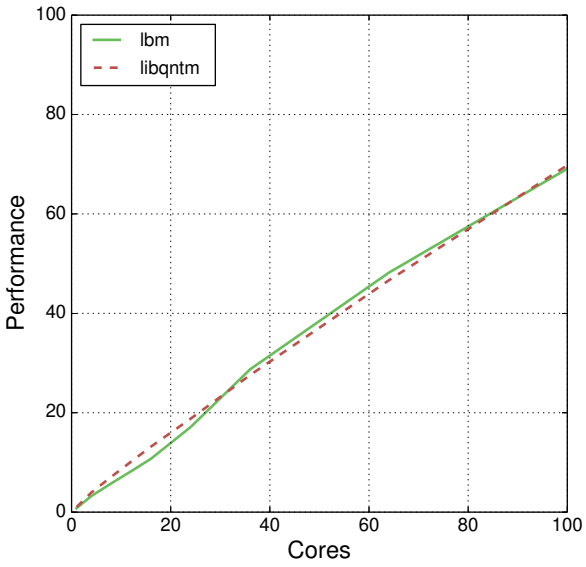


Figure 4-2: SCC's speedup over serial code up to 100 cores.

benchmarks by spawning loop iterations with highly parallel spawner trees. Figure 4-2 shows that linear scaling continues beyond 36 cores: SCC achieves 69× speedup for both `libquantum` and `lbm` on 100 cores.

SCC also finds significant parallelism in `milc`, by privatizing variables scoped within loops (Section 3.2.3) and composing parallelism across loops containing function calls. As a result, SCC scales `milc` to 12× at 36 cores.

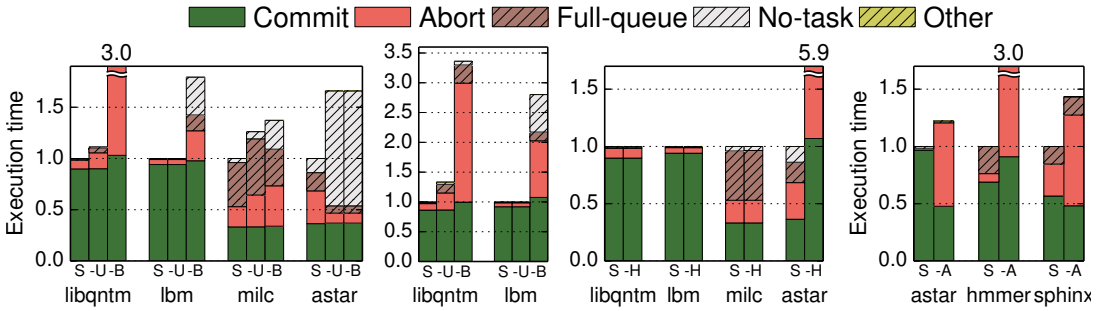
`astar` is parallelized as discussed in Section 3.8, yielding 5.7× speedup on 36 cores. To our knowledge, SCC is the first compiler to obtain any meaningful speedup over the serial version of `astar`. `hmmer` and `sphinx3` also use task-splitting annotations to reduce the impact of contentious data. SCC thus finds some parallelism in `hmmer` and `sphinx3`, achieving 1.6× and 1.9× speedups on 4 cores over the serial code, respectively.

Finally, SCC also finds parallelism in `mcf`. We add no annotations to the source code. A majority of time in `mcf` is spent in pointer-chasing loops, which SCC pipelines using chain expansion (Section 3.4.3). This automatically achieves 2.0× speedup on 4 cores with no programmer effort. Prior work had investigated an earlier version of `mcf` from SPEC CPU2000 and found that manually selecting good task boundaries in multiple hot code regions can achieve less than 3× overall speedup [7, 41]. These approaches are complementary, and future work could combine them to yield greater speedups.

4.2.3 Sensitivity studies

We study the effects of two transformations, loop expansion and spatial hint generation, which are key to the performance of some benchmarks that scale beyond 4 cores.

Analysis of loop expansion: To show the importance of SCC's balanced spawner trees, Figure 4-3a compares the execution time of SCC and SCC variants without progressive expansion for loops with unknown bounds (-U) or with spawner trees disabled for all loops (-B). Loops for which spawner trees are disabled use chain expansion instead. These results show that spawner trees always help and are often crucial for performance. `astar`'s execution time increases by 66% at 36 cores without progressive expansion. Meanwhile, although we saw spawner trees allowed `libquantum` and `lbm` to scale linearly to 100 cores, Figure 4-3b shows `libquantum` and `lbm` scale



(a) Loop expansion at 36 and **(b)** 100 cores. **(c)** Hints at 36 cores. **(d)** Task-splitting at 4 cores.

Figure 4-3: Normalized execution time for SCC when disabling different features: (a,b) disabling spawner trees for unbounded loops (-U) or all loops (-B), and (c) disabling spatial hint generation (-H). (d) Execution time increases when removing task-splitting annotations (-A).

much worse without spawner trees, and their performance at 100 cores drops by 3.4× and 2.8×, respectively.

Analysis of locality-aware execution: Figure 4-3c shows the effect of disabling spatial hint generation, which causes tasks that formerly used a spatial hint to be sent to a random tile. Locality-aware execution is critical to scale `astar`. We expect locality-aware execution to be more beneficial as the system scales, as shown in prior work [26].

Analysis of task-splitting annotations: Figure 4-3d shows the performance impact of disabling code modifications that encourage the compiler to split tasks at finer granualrity (Section 3.7) for the three benchmarks that use them: `astar`, `hmmer`, and `sphinx3`. All three become overwhelmed with expensive aborts with code modifications removed, highlighting the importance of identifying accesses to contentious data structures and splitting them into fine-grain tasks. As shown by Table 4.2, minimal code modifications are required in each case.

Comparison with out-of-order cores: We find that SCC is complementary to superscalar out-of-order cores. Figure 4-4 shows the performance of our benchmarks on out-of-order (OoO) cores, on a common scale with Figure 4-1a. We replaced our in-order core model with a Haswell-like 4-wide superscalar OoO core. For fairness, we keep cores clocked at 2 GHz. The OoO core achieves gmean 2.4× speedup on the original serial code. On top of this, SCC delivers a gmean 2.1× speedup on 4 OoO cores, and gmean 5.9× on 36 OoO cores, which is a 14× speedup over the in-order

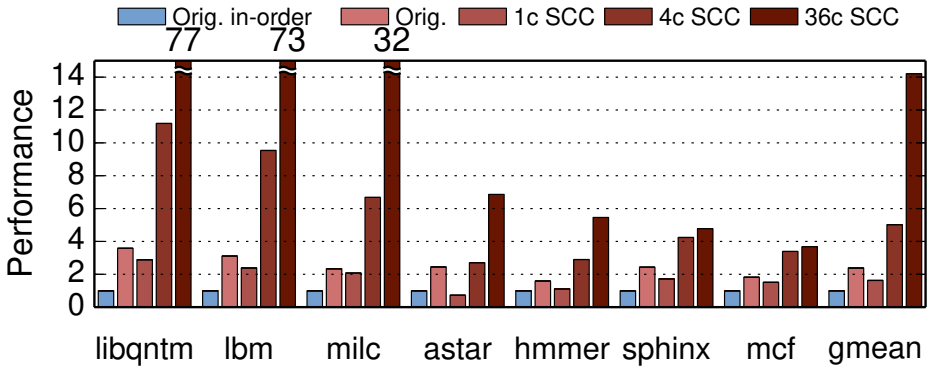


Figure 4-4: Performance of original serial code and SCC on 4-wide superscalar out-of-order cores, normalized to the original serial code on an in-order core from Figure 4-1a.

core baseline. This shows that design of SCC is orthogonal to the design of individual cores. Each core exploits ILP within the tasks SCC distributes to it, and SCC exploits ordered parallelism over far larger regions than OoO cores.

Chapter 5

Conclusion

We have presented SCC, a compiler that extracts fine-grain parallelism from sequential C/C++ programs. SCC exploits hardware features of the Swarm architecture with novel compiler transformations. It extracts fine-grain tasks to maximize parallelism, spawns tasks far in advance and out of order to keep cores busy, performs locality-aware execution, and optimizes tasks to reduce overheads, all while maintaining strictly deterministic and sequential semantics.

We use SCC to parallelize seven SPEC CPU2006 benchmarks, which are representative of real-world applications that are challenging to parallelize. We obtain a speedup of gmean 6.7× and up to 29× on 36 cores, and can scale even further, up to 69× on 100 cores. This demonstrates that speculative parallelization is profitable at larger scales and on a broader range of programs than shown in prior work. SCC also demonstrates that, with sufficient hardware support, it is possible to bring the performance benefits of parallelism to sequential programs. Future general-purpose processors that offer this hardware support will enable average programmers familiar with sequential programming to benefit from the increasing number of cores in their machines.

5.1 Future work

SCC raises interesting opportunities for future work. SCC's performance relies crucially on identifying accesses to contentious locations in memory and isolating them into

fine-grain tasks. In some benchmarks, we had to manually modify or annotate code to enable SCC to do this. These code modifications all retained strictly sequential semantics, so they could be performed by a programmer without needing to reason about a more complex parallel execution model. Nonetheless, it is appealing to consider future work that would fully automate the identification of contentious data and select good task structures without manual intervention. Possible approaches include using a profiling pass during compilation, using heuristics or machine learning to identify code patterns associated with contentious data accesses, or techniques to dynamically change the task structure of the code at runtime based on observed behavior.

SCC also paves the road to further research on software and hardware techniques to improve the performance of fine-grain speculative tasks. The Swarm architecture has so far used conventional general-purpose core designs, but it is appealing to adapt the core to optimize for the performance of such fine-grain tasks, and to further reduce task overheads. One approach to do this would be to more tightly integrate task spawn and dispatch mechanisms into the core pipeline, to allow overlapping the execution of multiple tasks. On the software side, further work can be done in reducing overheads by optimizing how closures are allocated in the heap. SCC's loop closure sharing amortizes the overhead of heap allocations across the iterations of a loop, but SCC would benefit from from a more holistic approach to analyzing the communication of values across tasks and amortizing their allocation overheads.

Another direction for research would be to explore what new primitives should be offered to programmers to enable programming patterns whose behavior can still be understood sequentially, but yield more efficient parallelized code. SCC's scalability is limited by contention in data structures and algorithms chosen by the programmer. Research into new programming languages, or high-level programming frameworks and libraries, could make it as easy for programmers to write code using data structures and algorithms amenable to scaling to tens or hundreds of cores with SCC's techniques and Swarm hardware support.

Finally, while so far SCC has focused on parallelizing single programs, further consideration needs to be given to building more complete systems, including system software and scheduling techniques to support speculatively parallelized multiprogram

workloads. If some or all of the programs running on a Swarm machine are parallelized at fine granularity, that presents new opportunities for system software working with hardware scheduling mechanisms to dynamically direct tasks to different cores at the time scale of microseconds, potentially better balancing the goals of throughput, fairness, and quality of service.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed., 2006.

[2] A. W. Appel, *Compiling with Continuations*. Cambridge university press, 1992.

[3] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *Proc. of the 3rd Intl. Symp. on Programming Language Implementation and Logic Programming*, 1991.

[4] Arvind, D. August, K. Pingali, D. Chiou, R. Sendag, and J. Y. Joshua, "Programming multicores: Do applications programmers need to write explicitly parallel programs?" *IEEE Micro*, no. 3, 2010.

[5] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *Proc. SPAA*, 2002.

[6] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *Proc. PLDI*, 2008.

[7] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proc. MICRO-40*, 2007.

[8] 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. ISCA-41*, 2014.

[9] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: automatic parallelization of irregular programs for chip multiprocessing," in *Proc. CGO*, 2012.

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

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

[12] 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. PPoPP*, 2003.

[13] 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. PLDI*, 2004.

[14] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM CSUR*, vol. 49, no. 2, 2016.

[15] J. Fix, N. P. Nagendra, S. Apostolakis, H. Zhang, S. Qiu, and D. I. August, "Hardware multithreaded transactions," in *Proc. ASPLOS-XXIII*, 2018.

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

[17] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors," in *Proc. HPCA-9*, 2003.

[18] 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.

[19] M. Halbherr, Y. Zhou, and C. F. Joerg, "MIMD-style parallel programming based on continuation-passing threads," in *Proc. of the 2nd Intl. Workshop on Massive Parallelism*, 1994.

[20] 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.

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

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

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

[24] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, 2006.

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

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

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

[28] 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.

[29] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez, "Harmonizing speculative and non-speculative execution in architectures for ordered parallelism," in *Proc. MICRO-51*, 2019.

[30] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-cut program decomposition for thread-level speculation," in *Proc. PLDI*, 2004.

[31] 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. PPoPP*, 2007.

[32] G. Kerneis and J. Chroboczek, "Continuation-passing C: compiling threads to events through continuations," *Higher-Order and Symbolic Computation*, vol. 24, no. 3, 2011.

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

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

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

[36] J. M. Martinez Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative POLyhedral Loop Optimizer," in *Proc. of the 7th International Workshop on Polyhedral Compilation Techniques*, 2017.

[37] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. MICRO-38*, 2005.

[38] V. Packirisamy, A. Zhai, W.-C. Hsu, P. C. Yew, and T. F. Ngai, "Exploring speculative parallelism in SPEC2006," in *Proc. ISPASS*, 2009.

[39] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, "Controlling program execution through binary instrumentation," *SIGARCH Comput. Archit. News*, vol. 33, no. 5, 2005.

[40] G. D. Plotkin, "Call-by-name, call-by-value and the λ-calculus," *Theoretical Computer Science*, vol. 1, 1975.

[41] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *Proc. PPoPP*, 2005.

[42] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen, "Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices," in *Proc. PLDI*, 2005.

[43] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August, "Speculative parallelization using software multi-threaded transactions," in *Proc. ASPLOS-XV*, 2010.

[44] R. Rangan, N. Vachharajani, G. Ottoni, and D. I. August, "Performance scalability of decoupled software pipelining," *ACM TACO*, vol. 5, no. 2, 2008.

[45] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proc. PACT-13*, 2004.

[46] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas, "Thread-level speculation on a CMP can be energy efficient," in *Proc. ICS'05*, 2005.

[47] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas, "Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation," in *Proc. ICS'05*, 2005.

[48] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *Proc. PPoPP*, 2017.

[49] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: The problem based benchmark suite," in *Proc. SPAA*, 2012.

[50] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proc. ISCA-22*, 1995.

[51] G. L. Steele Jr, "RABBIT: A compiler for SCHEME," Massachusetts Institute of Technology, Tech. Rep. AITR-474, 1978.

[52] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proc. ISCA-27*, 2000.

[53] J. G. Steffan and T. C. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," in *Proc. HPCA-4*, 1998.

[54] C. Strachey and C. P. Wadsworth, "Continuations: A mathematical semantics for handling full jumps," Oxford University Computing Laboratory, Tech. Rep. PRG-11, 1974.

[55] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *Proc. ISCA-44*, 2017.

[56] J.-Y. Tsai, Z. Jiang, and P.-C. Yew, "Compiler techniques for the superthreaded architectures," *Intl. Journal of Parallel Programming*, vol. 27, no. 1, 1999.

[57] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proc. PACT-16*, 2007.

[58] T. Vijaykumar and G. S. Sohi, "Task selection for a multiscalar processor," in *Proc. MICRO-31*, 1998.

[59] C. von Praun, L. Ceze, and C. Caşcaval, "Implicit parallelism with ordered transactions," in *Proc. PPoPP*, 2007.

[60] 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.

[61] J. Whaley and C. Kozyrakis, "Heuristics for profile-driven method-level speculative parallelization," in *Proc. 2005 Intl. Conf. on Parallel Processing*, 2005.

[62] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proc. HPCA-13*, 2007.

[63] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of scalar value communication between speculative threads," in *Proc. ASPLOS-X*, 2002.

[64] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler optimization of memory-resident value communication between speculative threads," in *Proc. CGO*, 2004.

[65] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors," in *Proc. HPCA-4*, 1998.