Appears in the Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA), 2020

T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware

Victor A. Ying
MIT CSAIL
victory@csail.mit.edu

Mark C. Jeffrey
*University of Toronto**
mcj@ece.utoronto.ca

Daniel Sanchez
MIT CSAIL
sanchez@csail.mit.edu

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

We present T4, a parallelizing compiler that successfully leverages recent hardware features for speculative execution, which present new opportunities and challenges for automatic parallelization. T4 transforms sequential programs into *trees of tiny timestamped tasks*. T4 introduces novel compiler techniques to expose parallelism aggressively across the entire program, breaking applications into tiny tasks of tens of instructions each. Task trees unfold their branches in parallel to enable high task-spawn throughput while exploiting selective aborts to recover from misspeculation cheaply. T4 exploits parallelism across function calls, loops, and loop nests; performs new transformations to reduce task spawn costs and avoid false sharing; and exploits data locality among fine-grain tasks. As a result, T4 scales several hard-to-parallelize SPEC CPU2006 benchmarks to tens of cores, on which prior work attained little or no speedup.

I. 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 [48]. Although some applications feature regular computations that are easy to parallelize, many programs feature data-dependent branches, imperative updates that use multiple levels of indirection, and codebases divided into many files and libraries. 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 such programs while retaining sequential semantics requires *speculative parallelization*. With speculative parallelization, the compiler divides 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).

*This work was done while Mark C. Jeffrey was at MIT.

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 [20, 24, 28, 63, 64, 68, 72, 77, 85]. TLS architectures suffered from three shortfalls that limited scalability: *(i)* resolving dependences by aborting all later tasks *en masse*, making aborts very expensive; *(ii)* one-task-at-a-time spawn or commit mechanisms, which bottleneck parallelism among tiny tasks; and *(iii)* lack of support for locality-aware execution.

Recent speculative architectures have proposed mechanisms that address the scalability limitations of TLS hardware [21, 32, 34, 35, 37, 74]. But these systems have been designed for explicitly parallelized code, and adapting TLS compiler techniques to these systems yields poor performance. These new architectures demand a new approach to automatic parallelization, and without it, their utility is limited.

To address this challenge we present *T4*, a compiler that speculatively parallelizes sequential programs to successfully leverage recent hardware for speculative parallelism. 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 scale:

- 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.
- 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.
- T4 spawns *tasks* with few inputs, not threads as in TLS. T4 carefully manages memory and register allocation to make task spawns very cheap, avoid false sharing, and eliminate the use of a shared call stack.
- T4 *exploits locality* by sending tasks that access the same data to the same tile in the system, reducing data movement.

To execute tiny ordered tasks efficiently, T4 targets the recent Swarm architecture [34, 35, 36, 74] (Sec. II). We implement T4 within the LLVM [47] compiler framework (Sec. III).

We evaluate T4 with ten SPEC CPU2006 C/C++ programs, which have frequent dependences that limited the effectiveness

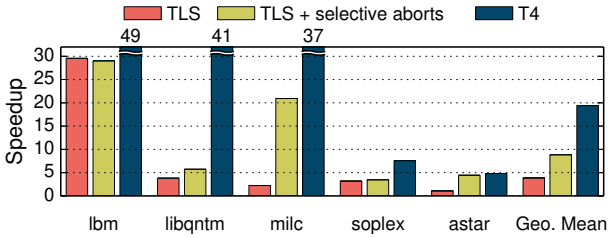


Fig. 1: 64-core performance of T4 significantly exceeds a TLS compiler based on prior work. Speedups are relative to serial code compiled with `-O3`. More details are in Sec. IX-C.

of prior work (Sec. IX). T4 broadens the range of applications that benefit from speculative parallelization and yields large scalability improvements over prior compiler techniques on several challenging applications. Fig. 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 sematics. T4 lets programmers unlock parallelism while retaining the simplicity of sequential programming. T4 and our hardware simulator are open source and publicly available at http://swarm.csail.mit.edu.

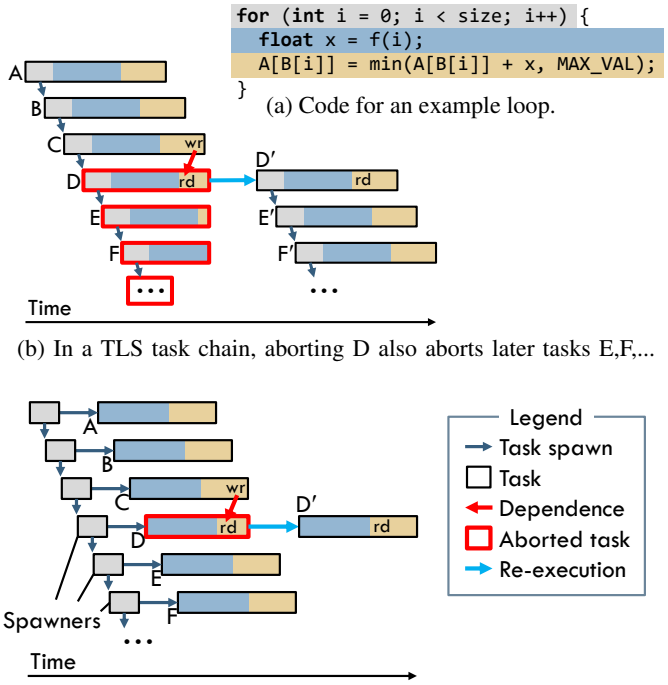
II. Trees of Tiny Timestamped Tasks

We now illustrate the limitations of prior TLS compilers and introduce key T4 features to overcome these limitations, explaining the large speedups in Fig. 1. We then detail the Swarm hardware baseline used to avoid scalability bottlenecks.

A. Spawner tasks enable selective aborts

Fig. 2a shows the code for an example loop that represents a common pattern found in many programs: each iteration first performs 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**. Fig. 2b 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 schedule tasks 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 Fig. 2b, task C writes to an address that conflicts with a read by task D. This conflict must be resolved by *aborting* D, and re-executing it (D′) so it is correctly ordered after C. Any later-ordered task that received incorrect data from D must also abort.



(a) Code for an example loop.

(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.

Fig. 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.

To implement this cascade of aborts, prior TLS architectures conservatively abort *all later tasks* en masse [24, 28, 64, 68, 72]. 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* [35]. 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 Fig. 2c. A chain of **gray** *spawner* tasks decouples spawns from work: each spawner spawns *(i)* the next spawner in the chain, then *(ii)* a separate *worker* task to execute the loop body. Now, D can abort without affecting later tasks. Spawners neither access nor suffer conflicts on application data, avoiding unnecessary abort cascades. Fig. 1 shows that selective aborts yields large speedups over TLS.

B. 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 Fig. 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

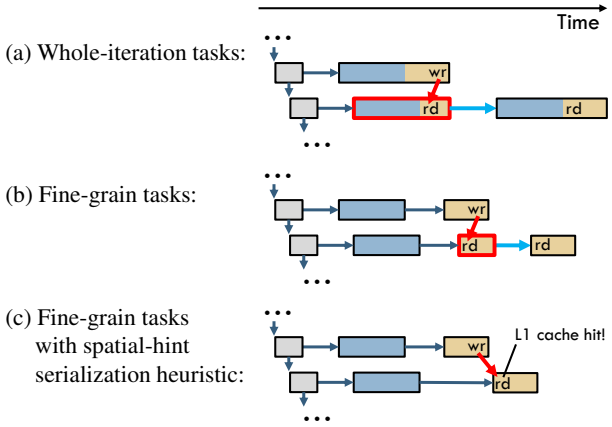


Fig. 3: Execution timelines depicting a data dependence through memory. Tiny tasks make aborts cheap and enable spatial hints.

only requires aborting and re-executing a cheap tiny task, as shown in Fig. 3b. If the function `f` 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 (Sec. II-D). This avoids ping-ponging of `A`'s cache lines across the chip. Furthermore, Swarm serializes the execution of same-hint tasks, avoiding aborts altogether, as shown in Fig. 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 about one task per cycle. Prior TLS systems could not achieve this throughput [35], 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 [64]. 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 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 [12, 21, 49, 60, 81].

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`.

C. Exponential trees make tiny tasks scale

Spawning tiny tasks quickly enough to keep many cores busy also requires new strategies in software. The serial chain of spawners in Fig. 2c 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 [42].

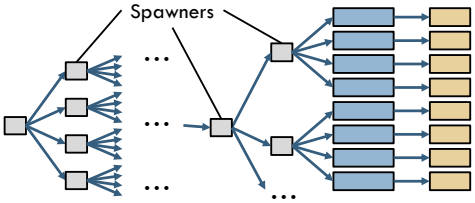


Fig. 4: Exponential spawner trees improve parallelism.

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

D. Baseline hardware architecture

T4 parallelizes sequential code by leveraging the Swarm architecture [35, 36]. Swarm extends a multicore that can run standard multithreaded programs with hardware support for speculative tasks. We use Swarm because it scales tasks as short as tens of instructions to hundreds of cores [34, 37, 74], but T4's techniques are general and would apply to any other system that performs efficient speculative execution of small ordered tasks. We first explain Swarm's execution model, then highlight the key microarchitectural features T4 must exploit to achieve good performance.

Swarm execution model: A Swarm program's output will always match a sequential model where execution is scheduled by a monotone priority queue. A Swarm program consists of tasks, where each task has a timestamp that acts as its key in the priority queue. Swarm guarantees that tasks appear to run in timestamp order, as if a single thread was popping the lowest-timestamp task from the priority queue and running it before popping the next task. Any task can spawn children tasks, inserting them into the priority queue with timestamps greater than or equal to the parent's timestamp. Swarm provides precise exceptions [37], so behavior conforms to the sequential model, even with arbitrary memory accesses or system calls.

Microarchitecture–What software needs to know: Swarm runs tasks speculatively and out-of-order to exploit parallelism. We focus on Swarm's key distributed hardware mechanisms that software should exploit to achieve good performance. For

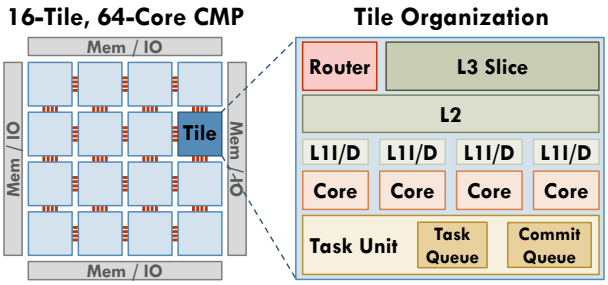


Fig. 5: 64-core/16-tile Swarm processor configuration.

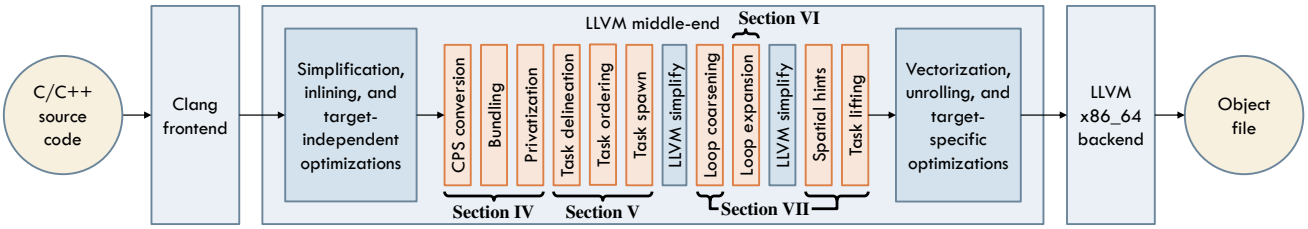


Fig. 6: Structure of T4, with corresponding sections of this paper. T4 is implemented by adding a series of passes to LLVM's middle-end. Transformation passes newly implemented for T4 are highlighted in orange.

Design goal	Hardware mechanism	Compiler transform
Composable order	Timestamps & domains	Topological task ordering
Avoid dependences	None needed	Stack elimination
Cheap aborts	Selective aborts	Spawners & tiny tasks
Cheap task spawn	Async. task spawns with register communication	Task lifting with live-in reduction
Parallel task spawn	Distributed task units	Progressive expansion
Exploit locality	Spatial hints	Spatial-hint generation

Table 1: To achieve most design goals, T4 introduces new program transformations to exploit hardware features.

hardware implementation details, please see prior work [35, 36, 74]. Table 1 summarizes how Swarm features couple with T4's compiler techniques for effective speculative parallelization.

Swarm extends prior mechanisms for speculation[1] to implement selective aborts, which allow recovery from misspeculation by aborting only small task subtrees or leaf tasks.

Rather than a centralized priority queue, Swarm has distributed task units that queue tasks near cores. For example, our implementation has one task unit in each four-core tile, as shown in Fig. 5. These task units add <2% area overhead to a standard multicore [35].

When a core spawns a new child task, the task is first buffered in the local task unit. Values for the new task are sent directly from registers, requiring the involvement of the core for just a few cycles. Subsequently, task units send tasks to each other using point-to-point messages, without involving cores or any central scheduler. A hardware *task queue* in each tile holds tasks waiting to run. Task queues are large (e.g., 256 tasks per tile), and they spill tasks to memory in the rare event that they fill up. Task queues prioritize the dispatch of tasks based on program order (timestamps), not spawn order, making it profitable to spawn tasks far in advance.

To exploit locality, each task can 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 task-spawn instruction. Hardware sends same-hint tasks to run in the same tile, by hashing each hint to get a destination tile ID [34]. Within a tile, hardware serializes same-hint tasks to avoid likely conflicts.

When a core finishes a task, the task unit holds the task's speculative state in a *commit queue* until the task commits. This lets cores execute other tasks while finished tasks wait to commit. Commit queues are large (e.g., 64 tasks per tile), so cores can run many tasks ahead of commit. To drain the commit queues, Swarm's high-throughput commit protocol uses hierarchical min-reductions to infer when tasks can commit [35, 36], allowing many tasks to commit per cycle [1].

As long as the task queues have tasks ready to run, cores do not wait for cross-chip communication latency. However, filling the system with enough *tiny* tasks can be challenging. T4's task trees achieve this with branches that unfold in parallel, spawning tasks fast enough to keep many cores busy.

III. T4 Implementation Overview

Swarm has previously been used to manually parallelize algorithms using *explicit* timestamps, e.g., to scale graph kernels that schedule tasks with a priority queue [35, 74]. Instead, T4 uses 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 [12, 21, 49, 60, 81], 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 [74] to compose parallelized code and perform speculative execution at the granularity of tiny tasks.

Fig. 6 gives an overview of T4's main components. T4 adds transformation passes to the LLVM/Clang compiler toolchain. All of T4's passes are *intraprocedural*, that is, the compiler performs them on one function at a time, without relying on expensive *inter*procedural 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.

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

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

Sec. V – 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.

Sec. VI – Loop expansion: T4 generates spawner trees for loops, exploiting Swarm's distributed task units for parallel task spawning. T4 can generate spawner trees even for loops with unknown tripcounts using *progressive expansion*.

[1] Swarm uses eager (undo-log-based) version management and eager (coherence-based) conflict detection with Bloom filters, like LogTM-SE [82].

Sec. VII – Reducing communication costs: T4 *coarsens* tasks with striding accesses patterns and generates spatial hints to exploit locality. Finally, T4 *lifts* tasks into separate LLVM IR functions, reducing register and memory accesses for task spawns, and often avoids any per-task memory allocations.

After this, the LLVM backend generates machine code.

IV. Eliminating the Call Stack

Before parallelizing the code, T4 first transforms it to reduce false sharing and eliminate the shared function call stack, which would otherwise become a point of contention among parallel tasks. To accomplish this, T4 introduces transformations that are safe for general, sequential code and do not rely on hardware support: *bundling* local variables into heap allocations and transforming functions with return values into *continuation-passing style* (CPS).

Bundling: T4 replaces local variables that would 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 thread-private stacks [71], but T4 does not use stacks to share values among tasks.

Privatization: When bundling variables, T4 also *privatizes* variables scoped within the body of a loop, allocating a separate instance for each iteration of the loop [53]. This avoids false dependences between loop iterations.

CPS conversion: Continuation-passing style conversion eliminates the stack as a record of function call frames and eliminates the notion of a function returning to its caller. This means there is no need to allocate memory on each function call to save stack frame pointers or return addresses.

T4 converts only functions with return values to CPS, by modifying them 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. CPS conversion modifies some callsites 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 Fig. 2a, 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.

Comparison with prior work: Cactus stacks as used in fork-join parallel languages [22, 25] 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 [4, 5, 69], where it is used as an intermediate representation that simplifies optimizations and eases code generation. We adapt CPS not to ease compilation but to eliminate contention on the stack during speculative parallelization. To the best of our knowledge, we are the first to automate CPS conversion for sequential C and C++.

V. Full-Program Fine-Grain Parallelization

To parallelize an entire program, T4 exploits the structure of loops and function calls to preserve program order with unbounded levels of nested task spawns.

A. Task delineation

T4 divides code into tasks by turning each loop iteration, loop continuation, function call, and function call continuation into a separate task. Like prior work, we find that this approach naturally limits tasks sizes to be fairly small [49, 79].

Consider again the pattern of execution demonstrated by Fig. 2a: 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 Fig. 2a, T4 automatically isolates the read-modify-write into a tiny task because it is a function call continuation.

B. 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 (Sec. II-B). These annotations affect performance, not program semantics: the program retains its sequential, deterministic behavior. Future work could use profiling to add fine-grain task boundaries automatically, based on identifying contentious variables that cause aborts.

C. Task ordering

T4 targets Fractal [74], which extends Swarm's execution model by placing tasks in a hierarchy of nested *domains*. Each task may create a subdomain and spawn children into that subdomain. Hardware will construct unique priority keys for each task so the domain and its creator execute as a single atomic unit. Within a domain, tasks are ordered with timestamps as before. Domains simplify the composition of separately parallelized code that uses independent timestamp schemes.

T4 orders tasks with a combination of timestamps and Fractal domains. This process starts with the *control flow graph* (CFG) of a function, a directed graph whose nodes are basic blocks. If the function will have internal tasks, T4 creates a domain for those tasks. Before assigning timestamps, T4 contracts loops into a single node, so that the remaining CFG becomes acyclic.[2] T4 then topologically sorts the nodes of the acyclic CFG, and assigns timestamps to tasks in topological order, treating each loop as a single task. Thus, it is guaranteed that these timestamps reflect program order.

For each loop, T4 creates a Fractal subdomain and repeats task delineation within the loop. To do this, T4 first gives each iteration a starting timestamp that is a multiple of the loop index. Then, T4 examines the CFG region made of the loop with the back edge removed, again using topological sorting with nested loops contracted, to assign timestamps reflecting task order within each iteration. T4 recursively repeats this process for nested loops, creating tasks at all loop nest levels.

D. 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 Fig. 2a, 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 (Sec. II-B). 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 (Sec. II-A).

VI. Parallel Loop Expansion

T4 adopts a multifaceted strategy to parallelize loops. Central to this strategy is the use of *spawners* as described in Sec. II. 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 branches of 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 [78]. To avoid flooding the machine with tasks, each spawner is timestamped according to the first loop iteration it will spawn, which prioritizes spawners properly to expand the tree gracefully.

[2] Node splitting can make all cycles into natural loops [33].

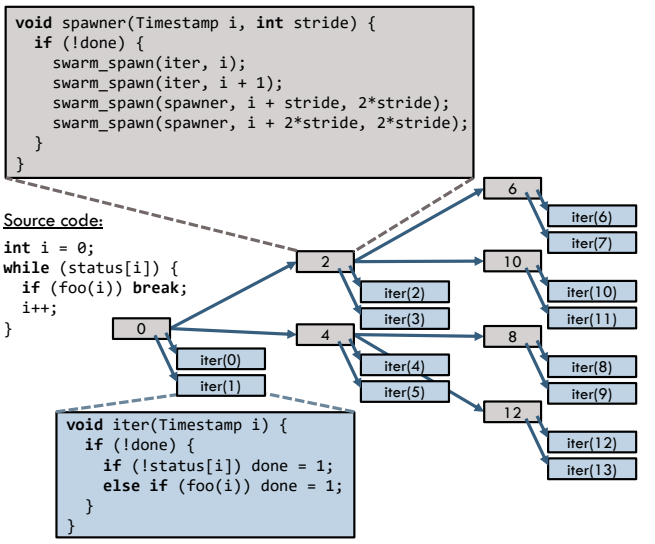


Fig. 7: Progressive expansion scales a `while` loop. Boxes show pseudocode for tasks of the transformed loop. The call to `foo` may spawn nested tasks (not shown).

A. Progressive tree expansion

Progressive expansion generates spawner trees for loops where the number of iterations is unknown. Fig. 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. Fig. 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 Fig. 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. In our evaluation, each spawner spawns four loop iterations and four child spawners, as we find this higher fanout improves scalability.

To handle unknown termination conditions, progressive expansion transforms control dependences into data dependences on a newly created `done` flag. Control-flow paths that would exit the loop write to this flag, as shown in the lower left of Fig. 7. Loop iteration and spawner tasks read this flag and exit early if the loop has already terminated. The new flag is an ordinary 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.

B. Bounded tree expansion

In some loops, LLVM analyses can find an expression for the number of iterations. For example, in Fig. 2a, the compiler knows the tripcount will be given at runtime by the variable `size`. For these *known-tripcount* loops, T4 can generate

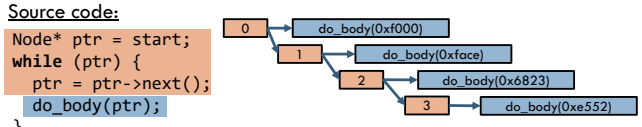


Fig. 8: Chain expansion avoids conflicts on `ptr`. Executions of `do_body` may spawn nested tasks in parallel (not shown).

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 Fig. 4. Our implementation of this compiler optimization for known-tripcount loops is similar to that in Tapir [66]. 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.

C. Chain expansion

While progressive expansion can operate on any loop, spawner trees are unprofitable for the loop in Fig. 8, where each iteration depends on the previous iteration's update of `ptr`. T4 identifies such *serializing* variables meeting three conditions: *(i)* every iteration unconditionally reads and writes to it, *(ii)* all computation in the loop body is dependent on the value of the variable, and *(iii)* the variable is not an induction variable that can be rewritten/eliminated. If all three conditions are met, T4 performs chain expansion. In practice, we find hot loops rarely require chain expansion.

As shown in Fig. 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, containing 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 [77].

VII. 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.

A. Loop task coarsening for cache alignment

To reduce false sharing and aborts, 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 [7, 46]. 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 [51], T4 coarsens loop tasks without relying on aliasing guarantees, eliminating false sharing due to striding accesses in many inner loops.

B. 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 (Sec. II-B). Sec. VIII presents a case study where this is crucial to obtain scalability.

C. Task lifting with live-in reduction

Up to now, T4 represented task code nested within larger functions, using an approach similar to Tapir [66]. 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.[3] 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 [2].
3) *Register packing* generates instructions to shift remaining live-ins 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.

VIII. 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`.

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

```
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.
     // ... more repetitions w/ different index1 ...
     // 8th of 8 identical code blocks appears here.
92 }
```

Listing 1: Code taken from Way_.cpp in `astar`, compressed and with repeated code blocks omitted

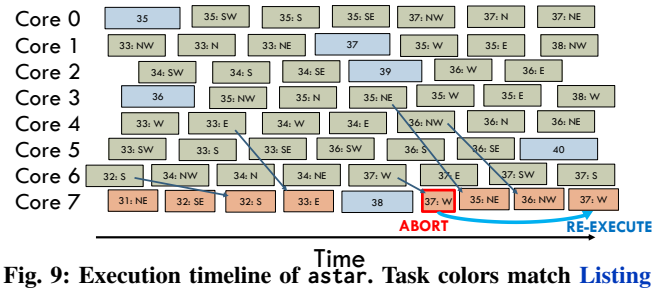


Fig. 9: Execution timeline of `astar`. Task colors match Listing 1. Numbers denote loop iterations. Arrows indicate spawns of neighbor-append tasks.

Listing 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 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 `bound2l`. T4 delineates tasks as shown in Fig. 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 (Sec. VII-B) 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 `bound2l` as a spatial hint, so accesses to this contentious variable execute serially in one tile. This is critical to achieve any speedup (Sec. IX-D). 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.

Cores	64 cores in 16 tiles (4 cores/tile), 3.5 GHz, x86-64 ISA; Haswell-like 4-wide OoO superscalar [27]
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
L3 cache	64 MB, shared, static NUCA [43] (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 (like Tile64 [80])
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 [14] 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
Spills	Spill 15 tasks when task queue is 86% full

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

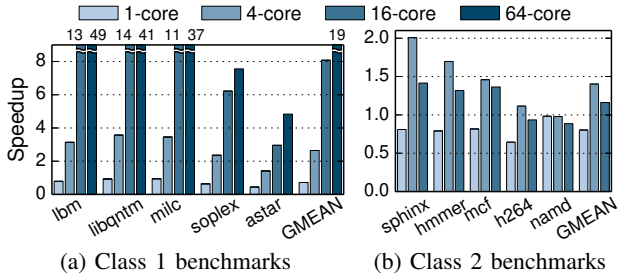
IX. Evaluation

A. Experimental methodology

Simulated hardware: We use a cycle-level simulator based on Pin [50, 57] to model Swarm systems with parameters given in Table 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 SPEC CPU2006 [29]. 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 auto-parallelize, even with speculation [19]. To the best of our knowledge, Packirisamy et al. obtained the best previous speedups for these benchmarks—1.6× at 4 cores, 1.91× at 8 cores—by combining



(a) Class 1 benchmarks (b) Class 2 benchmarks

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

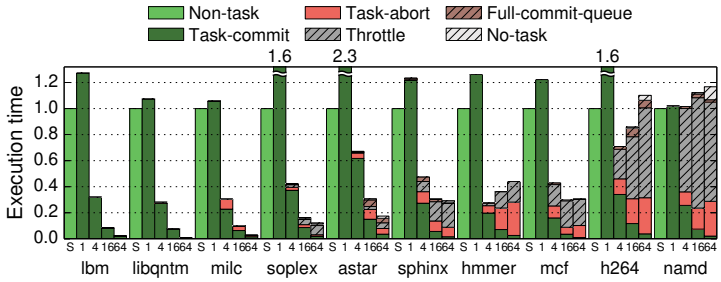


Fig. 11: Breakdown of execution time of T4 systems with 1, 4, 16, and 64 cores, normalized to the Serial code on a 1-core system.

advanced compiler techniques with hardware support for TLS and synchronization [56].

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`.[4] We evaluate all ten benchmarks that can be compiled by T4, listed in Table 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 [58, 67] 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 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 Sec. V-B. In `astar`, we deduplicate repetitive code and add annotations to break up tasks as shown in Sec. VIII. We manually perform loop fission in `hmmer` to allow T4 to better divide different striding memory accesses into parallel tasks that access separate cache lines (Sec. VII-A). 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 (Sec. IV).

We organize the evaluation as follows: we analyze T4's overall performance (Sec. IX-B), how T4's spawner trees are key to performance (Sec. IX-C), the effect of task-splitting annotations and hints (Sec. IX-D), task lifting optimizations (Sec. IX-E), and sensitivity to core microarchitecture (Sec. IX-F).

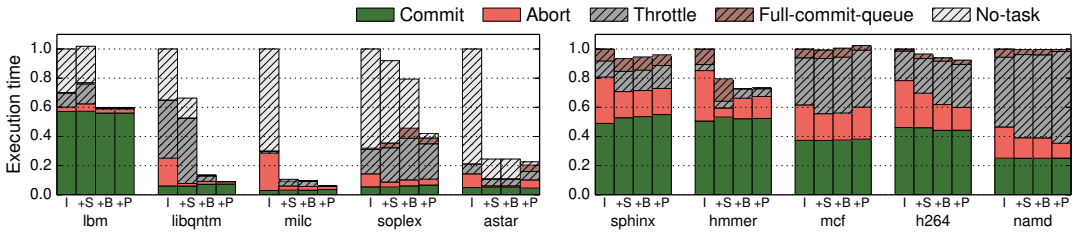
[4] Future work could address this, e.g., by passing escape continuations [4, 70]. This needs non-trivial engineering due to how exceptions are handled in LLVM.

B. T4 performance

Fig. 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× speedup. T4 scales multiple benchmarks to tens of cores, with up to 49× speedup on 64 cores (for `lbm`).

Fig. 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 (Sec. VII-C). 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. In return, short tasks make aborts cheap and confer high speedups.

Fig. 11 also shows a breakdown of how cycles are spent, averaged across cores. Cores execute *(i)* tasks that later commit or *(ii)* later abort, or they spend cycles idle because of *(iii)* the throttling heuristic, *(iv)* a full commit queue, or *(v)* no tasks 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 (Sec. II-D) 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 (Sec. VI) unfold trees gradually, so time lost to spilling task queues is <0.01%.



(a) Class 1 benchmarks at 64 cores

(b) Class 2 benchmarks at 4 cores

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

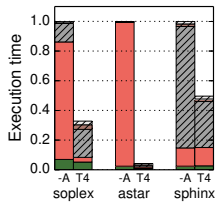


Fig. 13: 64-core execution time without (-A) and with (T4) task-splitting annotations.

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

T4 also finds significant parallelism in `soplex` and `astar`. yielding 7.6× and 4.8× 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 Fig. 2a. Meanwhile, `astar`'s inner loops are described in Sec. VIII. 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 (Sec. IX-D). 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 (Sec. VI-C), 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 [56] through aggressive compiler optimizations that move instructions off the critical path [83]. 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.

C. Benefits of spawners and spawner trees

Benchmarks show large variability in scalability, but T4's uses of spawners yield broad benefits across many benchmarks. Fig. 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. Fig. 12a shows 64-core results for Class 1 benchmarks (as in Fig. 1). Fig. 12b 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 [49, 64]. 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 (Sec. II-A). 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 (Sec. VI-B). 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.

D. Task-splitting annotations and spatial hints

Fig. 13 shows the performance impact of disabling code annotations that instruct T4 to split tasks at finer granularity (Sec. V-B) 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 (Sec. II-B). 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, 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× speedup for `astar` on any number of cores. This shows the importance of using tiny tasks that enable spatial hints (Sec. VII-B). Hints are critical to scalability, as otherwise a single frequently written cache line can cause copious aborts.

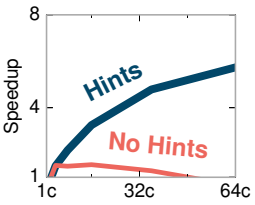


Fig. 14: `astar` needs spatial-hint generation.

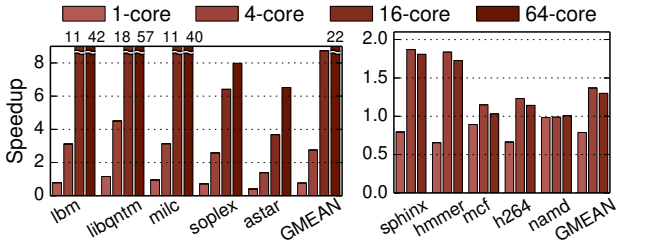
When `astar` repeatedly updates a single variable `bound2l`, these serialized operations form a critical path that limits whole-program performance (Sec. VIII). Fig. 14 shows that spatial-hint generation is critical to scale `astar`, and without it T4 would fall flat, achieving 1.5× speedup on 16 cores and deteriorating thereafter. Without spatial hints, tasks are sent to random tiles, ping-ponging the cache line in which `bound2l` resides.

E. 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 4 shows the volume of live-in values transferred from parents to children tasks, with and without loop environment sharing and live-in sinking (Sec. VII-C). 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 (Sec. II-D), much cheaper than prior TLS systems that moved whole register contexts. Table 4 shows that most of T4's task spawns do not allocate memory.

Benchmark	No optimizations bytes/task	No optimizations mem alloc	With optimizations bytes/task	With optimizations mem alloc
`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 4: T4's task lifting optimizations reduce NoC traffic due to task live-ins and reduce memory allocations for task spawns.



(a) Class 1 benchmarks (b) Class 2 benchmarks

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

F. 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, Fig. 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 Fig. 10. OoO cores run gmean 2.6× 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.

X. Related Work

Parallelizing sequential code is a long-standing problem for which many approaches have been tried. We 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 workloads. These software techniques could be combined with T4. Finally, we review prior work in TLS that combined hardware support and code transformations that carefully schedule tasks to extract parallelism from challenging applications with frequent dependences. T4 shares the goals of TLS but takes a different approach: dynamically scheduling tasks to exploit fine-grain parallelism and locality with mechanisms that avoid serial bottlenecks.

A. Non-speculative parallelizing compilers

Non-speculative parallelizing compilers [8, 53] divide sequential code into tasks that are guaranteed to be independent and can thus run in parallel. The key limitation of these compilers is that ensuring that two tasks are independent is often impossible at compile time. Polyhedral compilers [9, 26] can parallelize loops that perform regular accesses into arrays or matrices. But many programs are *irregular*: they use multiple levels of indirection or pointer-based structures, making static analyses ineffective [30, 41]. 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 has focused on exploiting *pipeline parallelism* in inner loops. DSWP [55, 61] pins loop iteration fragments across cores to localize loop-carried dependences, and relies on hardware

support for fine-grain inter-core communication. HELIX [13] implements efficient inter-thread communication in software, which suffices for some programs, and HELIX-RC [12] adds hardware support for inter-thread communication to accelerate a broader set of benchmarks. While these non-speculative parallelization techniques are highly efficient for the loops where they apply, they rely on static analyses to partition a loop into stages with unidirectional dependences, and are inapplicable if the loop contains occasional cyclic dependences. They use conservative serial execution for any code outside the loops they can parallelize and do not compose parallelism across nested loops or function calls.

B. Speculative parallelization in software

Like T4, some compilers leverage speculation to parallelize a broader range of sequential programs. Software-only speculative parallelization [3, 31, 38, 52, 60, 62] incurs significant overheads, especially to recover from misspeculation, so it is profitable only for applications 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 [39, 62]. Recent work uses profiling and static analysis to reduce these run-time overheads, but still suffers from expensive misspeculation recovery that makes it unprofitable for applications with frequent conflicts [3]. Future work could use these techniques to fully automate T4's privatization, as well as to extend privatization to objects in the heap where it is beneficial.

Salamanca et al. use strip-mining to avoid false sharing for loops with striding access patterns [65]. T4's loop task coarsening fully automates 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. Additionally, T4 uses spatial hints and stack elimination to reduce other aborts.

T4 uniquely uses Swarm's nested task spawns to exploit selective aborts. By using task boundaries as checkpoints to isolate contentious accesses, T4's spawners and tiny tasks repurpose Swarm's cheap task-spawn mechanisms to achieve benefits similar to alternative misspeculation recovery techniques that abort a portion of a task instead of an entire large task [16, 75]. García Yágüez et al. present a software-only system that performs selective aborts [23]. However, their system parallelizes one loop level, always aborts whole loop iterations, and does not support nested task spawns.

C. Thread-level speculation (TLS)

TLS architectures propose to use hardware mechanisms to make speculative parallelization more broadly beneficial for sequential programs [20, 24, 28, 63, 64, 68, 73, 85]. Unfortunately, TLS architectures suffer from expensive unselective aborts and serial spawn or commit mechanisms that cannot scale tiny tasks to many cores (Sec. II). Swarm addresses these issues, enabling T4's novel techniques.

Renau et al.'s TLS architecture [64] relaxes the requirement that task spawns must be serial, allowing speculative tasks to spawn children independently. This architecture can exploit some nested parallelism [49]. However, it does not decouple spawn order from execution order, so cores immediately execute tasks spawned early even if the tasks are very speculative and unlikely to commit. Moreover, it is still too restrictive to allow interleaving task timestamps as needed for progressive expansion. Finally, it performs serial commits that can bottleneck performance, as in other TLS architectures. To address this per-task overhead, it adaptively merges tasks if there are more tasks to run than cores. Thus, the system speculates at the coarsest granularity that fills the machine, resulting in large tasks prone to expensive unselective aborts. By contrast, Swarm's distributed queues manage many more tasks than cores, which T4 exploits for cheap selective aborts.

TLS compilers [6, 15, 18, 40, 56, 59, 76, 79, 83] are limited by the architectures they target. Like T4, studies on selecting tasks for the earliest TLS architectures consider function calls and loop iterations [54, 79]. However, without sufficient hardware support for tiny tasks, previous compilers have focused on *selectively* parallelizing coarser tasks. POSH [49] also spawns function calls and loop iterations, but it focuses on profiling to choose when *not* to use them as task boundaries, preferring to form sufficiently coarse tasks to amortize task overheads.

Many TLS compiler techniques have focused on parallelizing iterations from a single loop level. Many of these techniques limit the use of speculation to avoid the significant cost of unselective aborts. Du et al. propose models to statically estimate the likelihood of data dependences, to avoid parallelizing loops that could yield frequent aborts [18]. Zhai et al. develop compiler techniques to synchronize frequent data dependences instead of speculating on them [84]. They distribute loop iterations to cores in a rigid fashion to enable predictable point-to-point communication, making synchronization cheap. The synchronization forms a long *critical forwarding path* that includes every iteration of a loop. This rigid synchronization has some benefits and significant drawbacks over T4. On the one hand, for applications dominated by inner loops with dependent iterations (e.g., Class 2 benchmarks in Sec. IX), synchronization ensures work on the critical path runs in program order. By contrast, T4 suffers aborts due to order violations even with spatial hints (Sec. VIII). On the other hand, this rigid synchronization does not allow for nested task spawns and stalls cores when the work per iteration is imperfectly balanced [56], so cores spend time idle even if some iterations are independent (e.g., in Class 1 benchmarks). By contrast, T4 uses spatial hints to serialize only dependent tasks, running other tasks whenever available to keep tens of cores busy. Moreover, T4 parallelizes the whole program, composing parallelism across loop nests and functions calls. Future work is needed to combine the benefits of both approaches.

Kejariwal et al. show that task spawn overheads can eliminate speedup when spawning tiny tasks, such as from inner

loops [42]. SpecDSWP reduces loop parallelization overheads by a constant factor by pipelining loop fragments across cores so communication latency is off the critical path [77]. T4's chain expansion and spatial hints achieve similar benefits for loops that must communicate values from one iteration to another. T4's task trees yield asymptotically more parallelism by parallelizing task spawning itself, which delivers order-of-magnitude speedups for tiny tasks.

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 [49]. This impedes standard compiler optimizations [63], 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 [10, 44, 68]. T4's task-lifting optimizations would reduce the amount of data sent in such architectures, making task spawns cheaper for them just as it does for Swarm.

Some prior work proposes to exploit semantic commutativity with programmer support. This reduces aborts in speculative parallelization, but breaks sequential semantics, forcing programmers to reason with nondeterministic program output [11, 17, 37, 45]. By contrast, T4 always preserves sequential semantics, so programmers never worry about races.

XI. Conclusion

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.

Acknowledgments

We thank Elliott Forde for helping with spatial-hint generation, Grace Yin for experimenting with bundling and privatization, and Domenic Nutile for investigating performance bottlenecks. We thank Tao B. Schardl for many helpful discussions on compiler implementation. We thank Joel Emer, Po-An Tsai, Suvinay Subramanian, Guowei Zhang, Anurag Mukkara, Maleen Abeydeera, Quan Nguyen, Hyun Ryong (Ryan) Lee, Axel Feldmann, Yifan Yang, Keiko Yamaguchi, Shuichi Konami, and the anonymous reviewers for helpful discussions and feedback on this paper. This work was supported in part by a grant from Sony and by NSF grant CAREER-1452994. Mark C. Jeffrey was partially supported by a Facebook Fellowship.

References

[1] M. Abeydeera and D. Sanchez, "Chronos: Efficient speculative parallelism for accelerators," in *Proc. ASPLOS-XXV*, 2020.
[2] I. Akturk and U. R. Karpuzcu, "AMNESIAC: Amnesic automatic computer," in *Proc. ASPLOS-XXII*, 2017.
[3] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proc. ASPLOS-XXV*, 2020.
[4] A. W. Appel, *Compiling with Continuations*. Cambridge university press, 1992.
[5] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *Proc. PLILP*, 1991.
[6] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *Proc. SPAA*, 2002.
[7] 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.
[8] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence, "Parallel programming with Polaris," *Computer*, vol. 29, no. 12, 1996.
[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *Proc. PLDI*, 2008.
[10] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "The anatomy of the register file in a multiscalar processor," in *Proc. MICRO-27*, 1994.
[11] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *Proc. MICRO-40*, 2007.
[12] 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.
[13] 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.
[14] J. L. Carter and M. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. STOC-9*, 1977.
[15] 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.
[16] C. B. Colohan, A. Ailamaki, J. G. Steffan, and T. C. Mowry, "Tolerating dependences between large speculative threads via sub-threads," in *Proc. ISCA-33*, 2006.
[17] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *Proc. ASPLOS-XXIII*, 2018.
[18] 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.
[19] T. J. K. Edler von Koch and B. Franke, "Limits of region-based dynamic binary parallelization," in *Proc. VEE*, 2013.
[20] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A survey on thread-level speculation techniques," *ACM CSUR*, vol. 49, no. 2, 2016.
[21] J. Fix, N. P. Nagendra, S. Apostolakis, H. Zhang, S. Qiu, and D. I. August, "Hardware multithreaded transactions," in *Proc. ASPLOS-XXIII*, 2018.
[22] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proc. PLDI*, 1998.
[23] A. García Yágüez, D. R. Llanos, and A. González-Escribano, "Squashing alternatives for software-based speculative parallelization," *IEEE Trans. Computers*, vol. 63, no. 7, 2014.
[24] 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.
[25] 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.
[26] 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.

[27] P. Hammarlund *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, 2014.

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

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

[30] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *Proc. PASTE*, 2001.

[31] 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.

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

[33] J. Janssen and H. Corporaal, "Making graphs reducible with controlled node splitting," *ACM TOPLAS*, vol. 19, no. 6, 1997.

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

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

[36] 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.

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

[38] 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.

[39] N. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, "Speculative separation for privatization and reductions," in *Proc. PLDI*, 2012.

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

[41] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM CSUR*, vol. 49, no. 2, 2016.

[42] A. Kejariwal *et al.*, "Tight analysis of the performance potential of thread speculation using SPEC CPU2006," in *Proc. PPoPP*, 2007.

[43] 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.

[44] V. Krishnan and J. Torrellas, "The need for fast communication in hardware-based speculative chip multiprocessors," in *Proc. PACT-8*, 1999.

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

[46] S. Larsen, E. Witchel, and S. Amarasinghe, "Increasing and detecting memory address congruence," in *Proc. PACT-11*, 2002.

[47] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004.

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

[49] 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.

[50] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, 2005.

[51] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Proc. PACT-20*, 2011.

[52] J. M. Martinez Caamaño, A. Sukumaran-Rajam, A. Baloian, M. Selva, and P. Clauss, "APOLLO: Automatic speculative POLyhedral Loop Optimizer," in *Proc. IMPACT*, 2017.

[53] S. P. Midkiff, "Automatic parallelization: An overview of fundamental compiler techniques," *Synthesis Lectures on Computer Architecture*, 2012.

[54] J. T. Oplinger, D. L. Heine, and M. S. Lam, "In search of speculative thread-level parallelism," in *Proc. PACT-8*, 1999.

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

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

[57] 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.

[58] 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. MICRO-37*, 2004.

[59] 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.

[60] 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.

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

[62] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proc. PLDI*, 1995.

[63] 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.

[64] 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.

[65] J. Salamanca, J. N. Amaral, and G. Araujo, "Evaluating and improving thread-level speculation in hardware transactional memories," in *Proc. IPDPS*, 2016.

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

[67] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-X*, 2002.

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

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

[70] G. L. Steele Jr and G. J. Sussman, "Lambda: The Ultimate Imperative," Massachusetts Institute of Technology, Tech. Rep. AIM-353, 1976.

[71] 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.

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

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

[74] 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.

[75] C. Tian, C. Lin, M. Feng, and R. Gupta, "Enhanced speculative parallelization via incremental recovery," in *Proc. PPoPP*, 2011.

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

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

[78] R. A. van Engelen, "Efficient symbolic analysis for optimizing compilers," in *Proc. CC*, 2001.

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

[80] D. Wentzlaff *et al.*, "On-chip interconnection architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, 2007.

[81] J. Whaley and C. Kozyrakis, "Heuristics for profile-driven method-level speculative parallelization," in *Proc. ICPP*, 2005.

[82] 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.

[83] 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.

[84] 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.

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