Appears in the Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018

Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism

Mark C. Jeffrey* Victor A. Ying* Suvinay Subramanian* Hyun Ryong Lee* Joel Emer*† Daniel Sanchez*

*Massachusetts Institute of Technology †NVIDIA

{mcj, victory, suvinay, hrlee, emer, sanchez}@csail.mit.edu

***Abstract*—Multicore systems should support both speculative and non-speculative parallelism. Speculative parallelism is easy to use and is crucial to scale many challenging applications, while non-speculative parallelism is more efficient and allows parallel irrevocable actions (e.g., parallel I/O). Unfortunately, prior techniques are far from this goal. Hardware transactional memory (HTM) systems support speculative (transactional) and non-speculative (non-transactional) work, but lack coordination mechanisms between the two, and are limited to unordered parallelism. Prior work has extended HTMs to avoid the limitations of speculative execution, e.g., through escape actions and open-nested transactions. But these mechanisms are incompatible with systems that exploit ordered parallelism, which parallelize a broader range of applications and are easier to use.**

We contribute two techniques that enable seamlessly composing and coordinating speculative and non-speculative work in the context of ordered parallelism: *(i)* a task-based execution model that efficiently coordinates concurrent speculative and non-speculative ordered tasks, allowing them to create tasks of either kind and to operate on shared data; and *(ii)* a safe way for speculative tasks to invoke software-managed speculative actions that avoid hardware version management and conflict detection. These contributions improve efficiency and enable new capabilities. Across several benchmarks, they allow the system to dynamically choose whether to execute tasks speculatively or non-speculatively, avoid needless conflicts among speculative tasks, and allow speculative tasks to safely invoke irrevocable actions.

***Index Terms*—multicore, speculative parallelism, ordered parallelism, fine-grain parallelism, transactional memory, thread-level speculation, speculative forwarding, synchronization.**

I. Introduction

Systems that support speculative parallelism, such as thread-level speculation (TLS) and transactional memory (TM), have two major benefits over non-speculative systems: they simplify parallel programming [69, 79] and uncover abundant parallelism in many hard-to-parallelize applications [47, 89]. However, even applications that need speculation to scale have work that is best executed non-speculatively. For example, some tasks are well synchronized and running them speculatively adds overhead and needless aborts. Moreover, non-speculative parallelism is needed to perform irrevocable actions, such as I/O, in parallel.

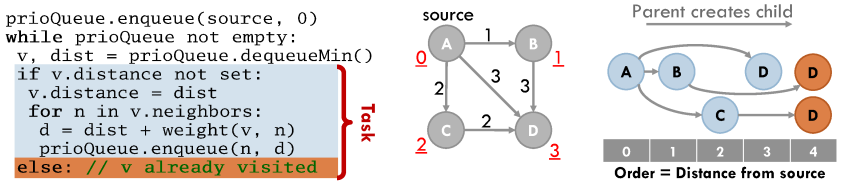
Ideally, systems should support composition and coordination of speculative and non-speculative tasks, and allow those tasks to share data. Unfortunately, prior techniques fall short of this goal. All prior hardware techniques to combine speculative and non-speculative parallelism have been done in hardware transactional memory (HTM) systems [16, 36, 41, 62, 73]. HTM supports both speculative (transactional) and non-speculative (non-transactional) code. But HTM *lacks shared synchronization mechanisms*, so speculative and non-speculative code cannot easily access shared data [26, 92]. Moreover, most HTMs provide *unordered* execution semantics that miss many opportunities for parallelization.

Recent work has instead focused on using speculation to support *ordered parallelism*, where parallel tasks appear to execute in a program-specified total or partial order [47, 70]. Ordered parallelism is more general and abundant than unordered parallelism. For example, consider the problem of parallelizing a transactional database. Using classic HTM, each database transaction must execute on a single thread as a long memory transaction. By contrast, ordered parallelism allows breaking each database transaction into many short, ordered tasks, exploiting abundant intra-transaction parallelism [29, 89].

Classic TLS systems leveraged ordered speculation to parallelize sequential programs [27, 30, 35, 75, 76, 86, 87, 97], some HTMs offer programmer-defined commit order [16, 36, 71], and the recent Swarm architecture [46, 47, 48, 89] has a rich execution model that can parallelize more algorithms than TLS or TM and supports tiny ordered tasks efficiently. However, *these systems disallow non-speculative parallelism*: all tasks except the earliest active one execute speculatively.

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

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



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

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

(c) Tasks executed by `sssp`. Each task shows the vertex it visits.

Fig. 1. Dijkstra's single-source shortest paths algorithm (`sssp`).

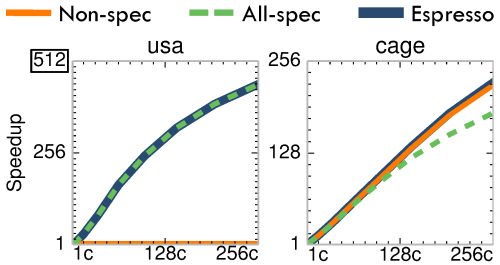


Fig. 2. Speedup of three versions of `sssp` on 1–256 cores for two graphs. Total cache and queue capacities grow with core count (Sec. VI), causing superlinear speedup in `usa`.

Our second contribution is *Capsules*, a technique that lets speculative tasks avoid hardware-managed speculation, enabling scalable system services and concurrent system calls (Sec. IV). Prior work in HTM has proposed escape actions [13, 63, 100] and open-nested transactions [59, 63, 64] to achieve similar goals. Unfortunately, these mechanisms are incompatible with many modern speculative systems. Specifically, they are incompatible with *speculative forwarding*, which allows speculative tasks to read data written by uncommitted tasks. Forwarding is critical for ordered parallelism, but causes tasks to lose data and control-flow integrity (Sec. II-C). Capsules solve this problem by implementing a safe mechanism to transition out of hardware-managed speculation and by protecting certain memory regions from speculative accesses. Unlike prior techniques, Capsules can be applied to any speculative system, even if speculative tasks can lose data or control-flow integrity.

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

II. Background and Motivation

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

A. Speculation benefits are input-dependent

Dijkstra's algorithm for single-source shortest paths (`sssp`) aptly illustrates the tradeoffs between speculative and non-speculative parallelism. `sssp` finds the shortest distance between some source vertex and all other vertices in a graph with weighted edges. Each task visits one vertex, and tasks execute in order of distance from the source. The first task to visit a given vertex sets its distance and creates tasks to visit all its neighbors; later tasks that visit the same vertex do nothing. Fig. 1 shows code for sequential `sssp`, which uses a priority queue to order tasks, and illustrates how it works.

`sssp` admits a non-speculative, one-distance-at-a-time parallelization [24, 55, 60]. At any given time, the system only processes tasks with the lowest unprocessed distance; these create tasks with higher distances. After all tasks for the current distance have finished, cores wait at a barrier and collectively move on to the next unprocessed distance. Since multiple same-distance tasks may visit the same vertex, each task must use proper synchronization to ensure safety.

This non-speculative `sssp` works well if the graph is shallow and there are many vertices with the same distance to the source. However, weighted graphs often have very few vertices per distance, so non-speculative `sssp` will find little work between barriers. In this case, scaling `sssp` requires exploiting *ordered parallelism*, processing tasks across multiple distances simultaneously. While most tasks are independent, running dependent (same-vertex) tasks out of order will produce incorrect results. Hence, exploiting ordered parallelism requires speculative execution, running tasks out of order and committing them in order. The Swarm architecture can do this, but it runs all tasks (except the earliest active one) speculatively [47].

Neither strategy is always the best. Fig. 2 compares the speedups of the non-speculative (Non-spec) and fully speculative (All-spec) versions of `sssp` on two graphs: `usa`, a graph of Eastern U.S. roads, and `cage`, a graph arising from DNA electrophoresis. Both versions leverage Espresso's hardware-accelerated task scheduling and locality-aware execution (see Sec. VI-A for methodology). At 256 cores, on `usa` All-spec is 248× faster than Non-spec, which barely scales because there are few vertices per distance. This is consistent with prior work [46, 47]. By contrast, `cage` is a shallow unit-weight graph with about 36000 vertices per distance, so Non-spec outperforms All-spec by 21%, as it does not incur the overheads of speculative execution.

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

Espresso provides the mechanisms needed for this hybrid strategy (Sec. III). First, it provides two synchronization mechanisms, timestamps and locales, that have consistent semantics across speculative and non-speculative tasks. Second,

it lets the system choose whether to speculate or not, based on the amount of available parallelism. Fig. 2 shows that the Espresso version of `sssp` achieves the best performance on both graphs, because it only uses speculative parallelism when non-speculative parallelism is insufficient.

B. Combining speculative and non-speculative tasks

Even in applications that need speculative parallelization, some tasks are best run non-speculatively. Consider `des`, a discrete event simulator for digital circuits, which we adapt from Galois [40, 70]. Each `des` task evaluates the effects of toggling a gate input at a particular simulated time; if the gate's output toggles, the task creates new tasks for gates connected to this output. As with `sssp`, ordered speculation enables `des` to scale to hundreds of cores [46, 47].

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

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

The right strategy is to let each speculative simulation task launch a non-speculative logging task. A logging task runs only after its parent commits, avoiding mispeculation. If its parent aborts, it is discarded. Espresso enables this approach.

Fig. 3 compares the speedups of using speculative and non-speculative logging in `des`. Speculative logging causes needless aborts that limit `des`'s performance beyond 64 cores. At 256 cores, non-speculative logging is 4.1× faster.

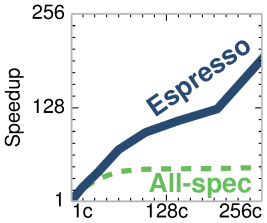


Fig. 3. `des` speedup on 1–256 cores, with speculative and non-speculative logging tasks.

Prior work in HTM has proposed to let transactions register *commit handlers* that run only at transaction commit [59]. Espresso generalizes this idea to let any task create speculative or non-speculative children. Additionally, Espresso's implementation requirements are different: unordered HTMs commit transactions immediately after running, while ordered tasks can stay speculative many cycles after they finish.

C. Software-managed speculation improves parallelism

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

Prior work in HTM has proposed *escape actions* for this purpose [13, 63, 100]. Escape actions let a transaction temporarily turn off hardware conflict detection and version management and run arbitrary code, including system calls. An escape action can register an abort handler that undoes its effects if the enclosing transaction aborts. For example, a transaction can use escape actions to allocate memory from a conventional thread-safe allocator, avoiding conflicts on allocator metadata. The escape action's abort handler frees the allocated memory.

Unfortunately, escape actions and similar mechanisms, such as open-nested transactions [59, 63, 64], are incompatible with architectures for ordered parallelism and many recent HTMs. These architectures perform *speculative forwarding* [4, 29, 44, 71, 72, 74, 86, 88], which lets tasks access data written by earlier, uncommitted tasks.[1] Speculative forwarding is crucial because ordered tasks may take a long time to commit. Without speculative forwarding, many ordered algorithms scale poorly [88] (e.g., `des` is 5× slower at 256 cores).[2] However, letting tasks access uncommitted state means they may read inconsistent data, and lose data and control-flow integrity (e.g., by dereferencing or performing an indirect jump to an invalid pointer). This makes escape actions unsafe: a mispeculating task can begin a malformed escape action, or an escape action might read temporarily clobbered data. Such an escape action could perform actions that cannot be undone by an abort handler.

Without escape actions, prior ordered speculative systems are forced to run memory allocation routines speculatively, and suffer from spurious conflicts on allocator metadata. Fig. 4 shows the performance of `des` when linked with TCMalloc [31], a state-of-the-art memory allocator that uses thread-local structures. `des` scales poorly with TCMalloc, achieving a speedup of 23× at 100 cores and declining significantly at higher core counts, where it is overwhelmed with aborts.

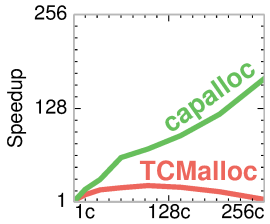


Fig. 4. `des` speedup on 1–256 cores, with different allocators.

Capsules (Sec. IV) solve this problem by providing a general solution for providing protected access to software-managed speculation. The program prespecifies a set of *capsule functions*, and the system guarantees that speculative tasks can only disable hardware speculation by invoking these functions. This prevents mispeculating tasks from invoking arbitrary actions. Moreover, capsule functions have access to memory that is not conflict-checked and is protected from accesses by speculative tasks. Fig. 4 shows the performance of `capalloc`, a memory allocator similar to TCMalloc that implements all allocation routines, such as `malloc`, in capsule functions and uses lock-based synchronization. `capalloc` makes `des` scale well, outperforming TCMalloc by 39× at 256 cores.

III. Espresso Execution Model

Espresso programs consist of tasks that run speculatively or non-speculatively. All tasks can access shared memory and make arbitrary system calls. Espresso provides two synchronization mechanisms: *timestamps* to convey order requirements and *locales* to convey mutual exclusion and locality information. Each task can optionally be given a timestamp and a locale.

[1] We follow TLS terminology [71, 86, 88]; in software TMs, lack of speculative forwarding is referred to as *opacity* [21, 34].

[2] Without speculative forwarding, systems must stall [62] or abort [11, 95] tasks that access uncommitted data.

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

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

Listing 1. Espresso implementation of Dijkstra's `sssp` algorithm.

Timestamps and locales have common semantics among speculative and non-speculative tasks, allowing tasks running in either *mode* to coordinate accesses to shared data.

Espresso supports three task *types* that control speculation: `SPEC` tasks always run speculatively, `NONSPEC` tasks always run non-speculatively, and `MAYSPEC` tasks may run in either mode. All tasks can create children tasks of any type.

We expose these features through a simple API. Tasks create children tasks by calling the following (inlined) function:

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

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

Espresso programs start by creating one or more initial tasks with `espresso::create` and calling `espresso::run`, which returns control after all tasks finish. Listing 1 shows the implementation of `sssp` described in Sec. II-A, which we will use to explain Espresso's semantics. In this example, the program creates one initial task to visit the source vertex.

A. Espresso semantics

Espresso runs all speculative tasks atomically, i.e., speculative tasks never appear to interleave. Moreover, Espresso provides strong atomicity [10, 20] between speculative and non-speculative tasks: the effects of a speculative task are invisible to non-speculative tasks until the speculative task commits (*containment* [10]), and non-speculative writes do not appear mid-speculative-task (*non-interference* [10]). Espresso does not guarantee atomicity among non-speculative tasks.

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

There are no atomicity guarantees among non-speculative tasks. The programmer must ensure non-`SPEC` tasks are *well-synchronized*, that is, they avoid race conditions.

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

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

TABLE I
THE EFFECT OF ESPRESSO'S SYNCHRONIZATION MECHANISMS.

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

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

Listing 1 shows timestamps in action. Each `sssp` task has a timestamp that corresponds to its path's distance to the source vertex. If all tasks had type `NONSPEC` instead of `MAYSPEC`, this code would implement the non-speculative, one-distance-at-a-time `sssp` version from Sec. II-A. If all tasks had type `SPEC`, the code would implement the fully speculative `sssp` version.

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

For non-speculative tasks, locales can be used as mutexes to write safe parallel code. For speculative tasks, locales are not necessary, since these tasks already appear to execute atomically. Locales are still useful in reducing aborts [46] as well as exploiting locality across speculative and non-speculative tasks.

Listing 1 shows locales in action. Each `sssp` task uses the ID of the vertex it processes as its locale. For non-speculative tasks, this implements mutual exclusion among tasks that access the same vertex. For both speculative and non-speculative tasks, this approach sends all tasks that operate on the same vertex to the same chip tile, improving temporal locality. An optimization to avoid cache ping-ponging could apply a technique from prior work [46]: use the cache line of the vertex as the locale.

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

Comparison with other execution models: Espresso generalizes both Swarm and HTM. Swarm programs consist of all-timestamped `SPEC` tasks. Espresso extends Swarm to support non-speculative tasks and to make timestamps optional.

Locales extend Swarm's spatial hints [46] to provide mutual exclusion among non-speculative tasks. Espresso also subsumes HTM. HTM programs consist of transactional (speculative) and non-transactional (non-speculative) code blocks. These are equivalent to non-timestamped `SPEC` and `NONSPEC` tasks.

B. `MAYSPEC`: *Tasks that may speculate*

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

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

Listing 1 shows `MAYSPEC` in action. All `sssp` tasks are tagged as `MAYSPEC` because they can run in either mode, as locales enforce mutual exclusion among same-vertex tasks. This implements the right strategy discussed in Sec. II-A: tasks with the lowest unprocessed distance run non-speculatively, and if this non-speculative parallelism is insufficient, the system runs higher-distance (i.e., higher-timestamp) tasks speculatively.

C. Exception model

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

To provide precise exception semantics and enforce strong atomicity, a speculative task that triggers an exception or a system call yields until it becomes the earliest active task and is then *promoted* to run non-speculatively. To guarantee promoted tasks still appear strongly atomic, a promoted task does not run concurrently with any other non-speculative task (Sec. V-B). Previous TLS systems used similar techniques to provide precise exceptions [35, 87].

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

Finally, Espresso introduces a `promote` instruction to expose this mechanism. If called from a speculative task, `promote` triggers an exception that will, in the absence of conflicts, eventually promote the task. If called from a non-speculative task, `promote` has no effect. `promote` has two uses. First, it can be invoked by tasks that detect an inconsistency and know they must abort, similar to transactional retry [38]. Second, `promote` lets code that must perform an expensive action avoid doing so speculatively (e.g., Listing 2 in Sec. IV).

IV. Capsules

Although hardware version management is more efficient than a software-only equivalent, hardware-only speculation can cause more serialization. For example, Espresso supports irrevocable actions in speculative tasks by promoting them to run non-speculatively, an expensive process that limits parallelism. Irrevocable actions cannot run under the control of hardware speculation, since hardware cannot undo their effects. This is limiting, because letting speculative tasks invoke system calls in parallel has many legitimate uses [6]. Beyond system calls, tasks may wish to perform software-managed speculative actions that exploit application-specific parallelization strategies, such as commutativity [18, 52, 65, 77]. As we saw in Sec. II-C, prior work proposed escape actions to achieve this goal [13, 63, 100]. But escape actions are incompatible with systems for ordered parallelism that need speculative forwarding. Forwarding makes speculative tasks lose data and control-flow integrity, making it impossible for software to dependably undo speculative actions.

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

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

A. Untracked memory

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

Software-managed speculative state should be maintained in untracked memory both to avoid the overhead of hardware

conflict detection as well as to ensure it is not corrupted by speculative tasks. Accesses to untracked data can be synchronized conventionally (e.g., with locks) to ensure safety.

B. Safely entering a capsule

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

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

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

C. Capsule execution

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

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

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

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

Listing 2. `malloc` implemented as a capsule function.

D. Capsule programming example

Listing 2 shows how `malloc` can be written as a capsule function. `malloc` first checks that its stack pointer is valid and has sufficient space, using `promote` otherwise. `malloc` also checks whether the requested allocation is very large, using `promote` if the program wants to allocate more than 16 MB. This avoids wasting excessive space to satisfy large requests from mispeculating tasks. After these checks, `malloc` calls `do_alloc`, which allocates the requested chunk. Finally, `malloc` uses `capsule_abort_handler` to register a call to `do_dealloc` as the abort handler. In this example, `do_alloc` and `do_dealloc` are thread-safe functions that use conventional synchronization (e.g., locks) to perform allocation and deallocation of heap memory. All allocator metadata (e.g., free lists) are stored in untracked memory. If the calling task aborts, the call to `do_dealloc` runs, freeing the allocated memory.

V. Implementation

We implement Espresso and Capsules by extending Swarm [2, 46, 47, 48, 89], a recent architecture for speculative parallelization. We choose Swarm as a baseline because it already provides most of the mechanisms needed for Espresso: it efficiently supports fine-grain tasks, implements scalable ordered speculation using timestamps, and performs locality-aware execution [46]. However, Espresso could be implemented over classic TLS systems as well, and Capsules are a general technique that could be applied to any speculative system, including HTM, TLS, Swarm, or Espresso. We first present Swarm's main features (see prior work [46, 47] for details), then describe how they are extended to implement Espresso, and finally describe the implementation of Capsules.

A. Baseline Swarm microarchitecture

The Swarm microarchitecture introduces modest changes to a tiled, cache-coherent multicore, shown in Fig. 5. Each tile has a group of simple cores, each with its own private L1 cache. All cores in a tile share an L2 cache, and each tile has a slice of a fully shared L3 cache. Each tile is augmented with a *task unit* that queues, dispatches, and commits tasks.

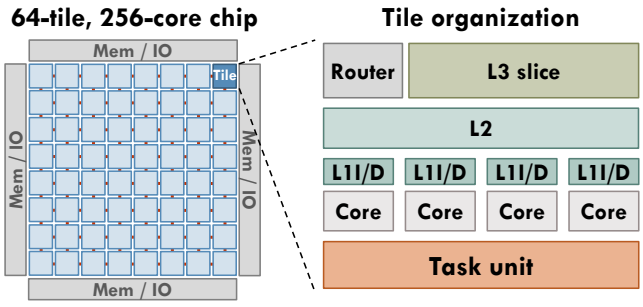


Fig. 5. 256-core chip and tile configuration.

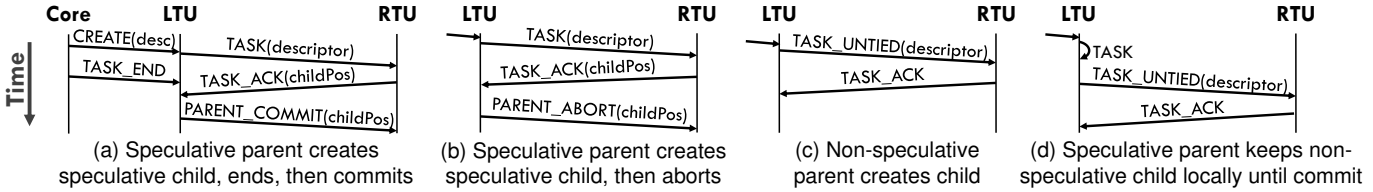


Fig. 6. Swarm (a,b) and Espresso (a,b,c,d) enqueue protocol between a local task unit (LTU) and a remote task unit (RTU), as a parent task creates a child.

Unlike Espresso, Swarm programs consist exclusively of speculative timestamped tasks. Swarm uncovers parallelism by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest active task. Swarm efficiently supports fine-grain tasks and a large speculation window through five techniques: hardware task management, large task queues, scalable speculation, high-throughput ordered commits, and locality-aware execution.

Hardware task management: Each task unit queues runnable tasks and stores the speculative state of finished tasks until they commit. Each task is represented by a task descriptor that contains its function pointer, timestamp, and arguments.

Tasks are created using a `create_task` instruction with arguments passed through registers. The local task unit asynchronously *enqueues* tasks to remote tiles. The parent's speculative state tracks where each child is enqueued. This enables parent commit or abort notifications to be sent to those children, as shown in Fig. 6(a) and Fig. 6(b). A parent abort notification aborts and discards the child, while a parent commit notification permits mechanisms to relieve queue pressure (see below).

To uncover enough parallelism, a task unit can dispatch any task to a core, even if its parent is still speculative. Cores dequeue tasks for execution by timestamp priority from the local task unit. A successful dequeue initiates speculative execution at the task's function pointer and makes the task's arguments available in registers. A core stalls if there is no task to dequeue.

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

Large task queues: The task unit has two main structures: *(i)* a *task queue* that holds task descriptors for every task in the tile, and *(ii)* a *commit queue* that holds the speculative state of tasks that have finished execution but cannot yet commit. Together, these queues implement a task-level reorder buffer.

Task and commit queues support tens of speculative tasks per core (e.g., 64 task queue entries and 16 commit queue entries per core) to implement a large window of speculation (e.g., 16 thousand tasks in the 256-core chip). Nevertheless, task and commit queues can fill up. This requires some simple actions to ensure forward progress. Specifically, tasks that receive parent-commit notifications can be *spilled* to memory to free task queue entries. If no tasks can be spilled, queue resource exhaustion is handled by either stalling task creation or aborting higher-timestamp tasks to free space [47].

Scalable speculation: Swarm enhances previously proposed speculation mechanisms to support a large number of speculative tasks. Swarm uses eager (undo-log-based) version management and eager conflict detection using Bloom filters, similar to LogTM-SE [94]. Swarm forwards still-speculative data read by a later task. When a task aborts, Swarm selectively aborts only its descendants and data-dependent tasks.

Swarm detects conflicts at cache line granularity, and leverages the cache hierarchy to substantially reduce the number of conflict checks and their cost [47]: L1 caches are managed so that L1 hits need not be conflict-checked, and L2 caches are managed so that L2 hits need only be conflict-checked against tasks in the same tile. The L3 directory maintains metadata so that L2 misses are conflict-checked only against tiles where uncommitted tasks may have accessed the same cache line.

To perform speculative forwarding and commits, Swarm dynamically produces a total order among tasks. Each task is given a unique *virtual time* (VT) when it is dispatched. VTs are 128-bit integers that extend each 64-bit programmer-assigned timestamp with a unique 64-bit *tiebreaker*. Swarm only allows tasks to access speculative data written by lower-VT tasks, and commits tasks in VT order to preserve correctness.

High-throughput ordered commits: Swarm adapts the virtual time algorithm [45] to achieve high commit throughput. Tiles periodically communicate with an arbiter (e.g., every 200 cycles) to discover the VT of the earliest (lowest-VT) active (unfinished) task in the system. All tasks with lower VTs can then commit. This scheme uses a hierarchical min reduction and can commit many tasks per cycle, thus scaling to hundreds of cores with tasks as short as a few instructions.

Locality-aware execution: Finally, Swarm leverages a technique called *spatial hints* [46] to perform locality-aware execution. A hint is an optional 64-bit integer that, much like a locale, abstractly denotes the data the task is likely to access.

Swarm exploits hints by running same-hint tasks in the same tile and serializing them. The hint is stored in the task's descriptor. When a core creates a new task, the task unit hashes its 64-bit hint to a tile ID, then enqueues the task to the selected tile. Thus, same-hint tasks run on the same tile. Tasks without hints are enqueued to random tiles. Then, the dispatch logic at each tile serializes same-hint tasks to avoid conflicts.

B. Espresso microarchitecture

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

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

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

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

A non-speculative task frees its task queue entry when dispatched and does not use a commit queue entry. This reduces queue pressure. Since the task can never abort, it does not track its children or send them any notifications, as shown in Fig. 6(c). This reduces traffic and allows non-speculative tasks to create an unbounded number of children; their children can always be spilled to memory.

Mixing non-speculative and speculative tasks requires changing the dispatch, conflict detection, and commit mechanisms:
1. Speculative `NONSPEC` *enqueue:* If a speculative task creates a `NONSPEC` child destined to a remote tile, since the child cannot run before its parent commits, the task is buffered locally and enqueued to the remote tile only when the parent commits (Fig. 6(d)). This avoids needless abort and commit notifications.
2. `MAYSPEC` *dispatch:* A `MAYSPEC` task is always speculatively runnable, but may exploit non-speculative execution for efficiency. The dispatch logic checks if the task meets the same conditions for a `NONSPEC` task to run. If so, the task executes non-speculatively; otherwise, it executes speculatively.
3. Conflicts: A non-speculative task does not track read/write-sets. However, to implement strong atomicity, its accesses are conflict-checked against speculative tasks. A non-speculative access that conflicts with a *running* speculative task's read/write set aborts the speculative task.

If a non-speculative task *N* conflicts with a *finished* speculative task *S*, *S* may be unsafe to abort. Recall that tiles periodically communicate to find the virtual time (VT) of the earliest active task, and all finished tasks whose VTs precede that earliest active VT must then commit. If the tile has sent a locally earliest active VT to the commit arbiter that is higher than *S*'s VT, *S* may be declared committed by the arbiter. To handle this race, *N* stalls until the arbiter replies and *S*'s fate is known. This race is very rare.
4. Commit protocol: When tiles send their earliest active VT to the arbiter, the timestamps of non-speculative tasks are included for consideration. This prevents any speculative task with a higher timestamp from committing, while allowing same-timestamp speculative tasks to commit.

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

Whereas TLS schemes stall the core running the exceptioned task [35, 87], Espresso leverages commit queues to avoid holding up a core. The *exceptioned* task is immediately stopped and its core becomes available to run another task. Its writes are rolled back, and its children tasks are aborted and discarded. Espresso then keeps the task's *read set* active in the commit queue. If the read set detects a conflict with an earlier task, the exceptioned task was mispeculating, so it becomes runnable again for *speculative* execution. However, if the task becomes the earliest active task without having suffered a conflict, it legitimately needs to perform an irrevocable action.

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

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

C. Capsules implementation

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

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

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

Safely entering a capsule: The *capsule-call vector* contains pointers to all capsule functions and is stored at a fixed location in untracked memory. It would be cumbersome to manually assign unique IDs to capsule functions and build the call vector. However, the linker and loader can automate this process, similarly to how they handle position-independent code [42].

Capsule aborts: If a task aborts, any registered abort handlers must run non-speculatively. After an abort, we enqueue each abort handler as a `NONSPEC` task with no timestamp.

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

VI. Evaluation

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

A. Experimental methodology

Modeled system: We use a cycle-level, execution-driven simulator based on Pin [57, 68] to model systems of up to 256 cores, as shown in Fig. 5, with parameters in Table II. Swarm parameters match those of prior work [46, 47, 48]. We use detailed core, cache, network, and main memory models, and simulate all task and speculation overheads (e.g., task traffic, running mispeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). Our 256-core configuration is similar to the Kalray MPPA [23]. We also simulate smaller systems with square meshes ($K \times K$ tiles for $K \leq 8$). Our 1-core system always runs all tasks non-speculatively. We keep *per-core* L2/L3 sizes and queue capacities constant across system sizes. This captures performance per unit area. As a result, larger systems have higher queue and cache capacities, which sometimes cause superlinear speedups.

TABLE II
Configuration of the 256-core system.

Cores	256 cores in 64 tiles (4 cores/tile), 2 GHz, x86-64 ISA; single-issue in-order, scoreboarded (stall-on-use) [46]
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 8-way, inclusive, 7-cycle latency
L3 cache	64 MB, shared, static NUCA [51] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	8×8 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [93])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (16384 total), 16 commit queue entries/core (4096 total)
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
Virtual 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 III
Benchmarks: source implementations and inputs; run time, average task length, and serial-relative performance on a single-core system.

Application	Input	1-core cycles total	1-core cycles per task	1-core perf. vs. serial
sssp [70]	cage14 [22]	1.6 B	53	0.93×
	East USA roads [1]	2.4 B	299	1.74×
cf [84, 98]	movielens-1m [37]	1.5 B	59500	0.98×
triangle [84]	R-MAT [17]	59.5 B	1240	1.02×
kmeans [61]	m40 n40 n16384 d24 c16	8.6 B	6500	1.02×
color [39]	netflix [8]	11.1 B	163	1.42×
bfs [55]	hugetric-00020 [5, 22]	3.3 B	139	0.93×
mis [85]	R-MAT [17]	1.7 B	121	0.80×
astar [47]	Germany roads [67]	1.6 B	458	1.37×
genome [61]	g4096 s48 n1048576	2.3 B	850	1.01×
des [70]	csaArray32	1.7 B	506	1.82×
nocsim [3]	16x16 mesh, tornado	19.3 B	979	1.79×
silo [90]	TPC-C, 4 whs, 1 Ktxns	0.1 B	3380	1.13×

Benchmarks: Table III reports the benchmarks and inputs used to evaluate Espresso and the Capsules-based allocator. We consider 17 ordered and unordered benchmarks.

We ported 15 benchmarks from Swarm [46, 47, 89], (all except those that need Fractal [89] to scale). Eight of the 15 benchmarks have tasks that can be well-synchronized with timestamps and locales: `sssp`, `color`, `bfs`, `mis`, `astar`, and `des` are ordered applications, and `genome` and `kmeans` are unordered transactional applications.

We also port bulk-synchronous `cf` (collaborative filtering) and `triangle` (triangle counting) from Ligra [84, 98]. Their tasks are well-synchronized: they perform lock-free atomic updates and use barriers, which we replace with timestamps.

Sec. VI-B compares Espresso versions by declaring all well-synchronized tasks as `SPEC`, `NONSPEC`, or `MAYSPEC`. The source code is otherwise identical. Swarm runs all tasks speculatively, and is thus equivalent to Espresso's `SPEC`. We also evaluate state-of-the-art software-only parallel versions as in prior work [46, 47, 89] (except for `genome` and `kmeans`, which lack a non-transactional parallel version, and `astar`, for which software-parallel versions yield no speedup [47]).

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

We report speedups relative to *tuned* 1-core Swarm implementations. Due to hardware task management, 1-core Swarm versions are competitive with (and often faster than) tuned software-only *serial* implementations, as shown in Table III.

We fast-forward each benchmark to the start of its parallel region and run the entire parallel region. We perform enough runs to achieve 95% confidence intervals $\leq 1\%$.

Memory allocation: Only two of the benchmarks used in Sec. VI-B (`genome` and `des`) allocate memory within tasks. To separate concerns, we study the impact of allocators in

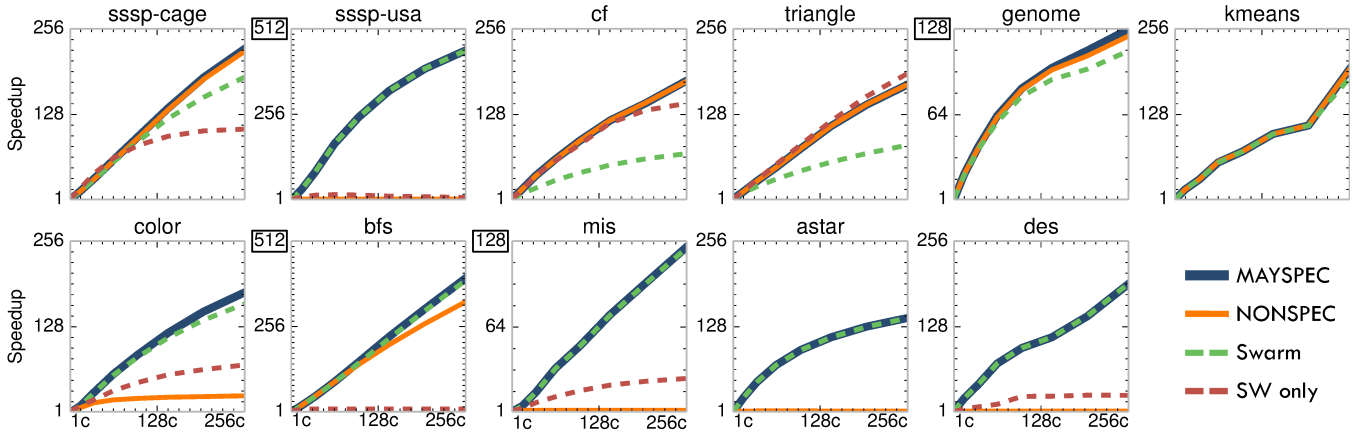


Fig. 7. Speedup of Swarm (SPEC), NONSPEC, MAYSPEC, and software-only benchmark variants on 1–256 cores. Higher is better.

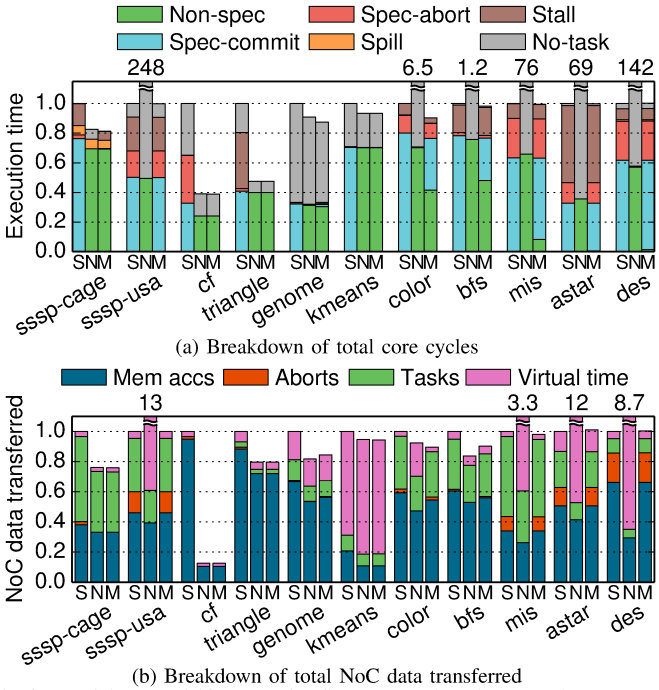
Sec. VI-C and use an ideal memory allocator in Sec. VI-B. The ideal allocator is implemented within the simulator, and allocates and deallocates heap memory from per-core pools with zero overhead. Memory freed by a speculative task is not reused until the task commits. For fairness, software-only implementations also use this allocator.

B. Espresso evaluation

Fig. 7 compares the performance of Swarm, Espresso's NONSPEC and MAYSPEC variants, and the software-only parallel versions as the system scales from 1 to 256 cores. Because most applications are hard to parallelize, MAYSPEC always matches or outperforms the software-only versions, which scale poorly in all cases except sssp-cage, cf, triangle, and color. Thus, we do not consider software-only versions further. Among the other schemes, Swarm works poorly on cf and triangle, and sacrifices some performance on sssp-cage, genome, and color; NONSPEC scales well in sssp-cage, cf, triangle, genome, kmeans, and bfs, but performs poorly in other applications because it forgoes opportunities to exploit speculative parallelism; and MAYSPEC always performs best.

Fig. 8 gives more insight into these results by showing core cycle and network traffic breakdowns at 256 cores for the Swarm, NONSPEC, and MAYSPEC versions. Each group of bars shows breakdowns for a different application. The height of a bar in Fig. 8(a) is the execution time relative to Swarm. Each bar shows a breakdown of how cores spend these cycles, executing *(i)* non-speculative tasks, or *(ii)* speculative tasks that later commit or *(iii)* later abort; *(iv)* spilling tasks to memory; *(v)* stalled on a full task or commit queue; or *(vi)* idle because there are no tasks available to run. Each bar of Fig. 8(b) reports the total bytes injected into the NoC relative to Swarm, broken down into four categories: *(i)* memory accesses from running tasks (between L2s and L3, or L3 and main memory), *(ii)* abort traffic (parent abort notifications and rollback memory accesses), *(iii)* task enqueues and parent commit notifications, *(iv)* virtual time updates (for ordered commits and barriers).

For sssp, as discussed in Sec. II-A, neither Swarm nor NONSPEC perform best across inputs. MAYSPEC outperforms the best of Swarm and NONSPEC by using speculation op-



(a) Breakdown of total core cycles

(b) Breakdown of total NoC data transferred

Fig. 8. Breakdowns at 256 cores, using Swarm (SPEC), and Espresso's NONSPEC, and MAYSPEC variants. Each bar is normalized to Swarm. Lower is better.

portunistically. In the shallow graph (cage), MAYSPEC runs almost all tasks non-speculatively. Meanwhile, in the deep graph (usa), MAYSPEC runs almost all tasks speculatively, overlapping the processing of vertices at multiple distances to extract enough parallelism. NONSPEC and MAYSPEC spend fewer cycles executing tasks than Swarm's committed cycles because non-speculative execution is more efficient: it reduces cache pressure (no undo log) and network traffic (less cache pressure, no aborts, and no parent commit notifications).

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

and have up to 8.0× (cf) lower network traffic.

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

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

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

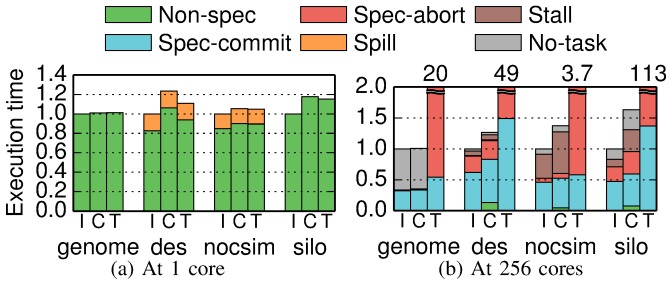
C. Capsules case study: Dynamic memory allocation

We design a scalable speculation-friendly memory allocator using Capsules. As discussed in Sec. II-C, simply calling memory allocation routines within speculative tasks introduces needless conflicts that limit parallelism. Prior work has proposed allocators for software TM [43], and used HTM to accelerate allocators *for non-speculative parallel programs* [25, 54]. TMs without forwarding can avoid false conflicts on allocator metadata by using escape actions or open-nested transactions [100], but as far as we know, no memory allocator has been implemented for systems with speculative forwarding.

To this end, we implement capalloc, a memory allocator built with Capsules. All allocation calls (malloc, calloc, etc.) are capsule functions, implemented following the pattern in Listing 2. To avoid reusing memory freed by tasks that later abort, each deallocation call (free, cfree) creates a non-speculative child with no timestamp to perform the deallocation. Thus, memory is deallocated only after the caller commits.

capalloc's internal design mimics TCMalloc [31], a state-of-the-art and widely used memory allocator. Small allocations (≤16 KB in our implementation) are served by a per-core software cache. These caches hold a limited amount of memory, and allocate from a set of central freelists. Large allocations are served from a centralized page heap that prioritizes space efficiency. The central freelists, large heap, and system-wide page allocator use spinlocks to avoid data races.

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



(a) At 1 core (b) At 256 cores

Fig. 9. Normalized execution time using three implementations of dynamic memory allocation: Ideal, Capalloc, and unmodified TCMalloc. Lower is better.

As explained in Sec. VI-A, we evaluate capalloc on the four Swarm applications with frequent dynamic allocation. We compare capalloc with TCMalloc and the ideal allocator.

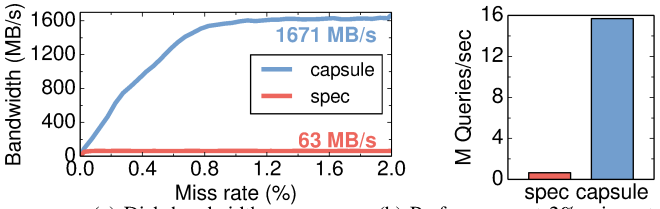
Fig. 9(a) shows single-core results, which let us examine work efficiency without concern for parallelism. Each group of bars shows execution times for one application, normalized to the ideal allocator. capalloc and TCMalloc perform similarly, adding gmean slowdowns of 11% and 8%, respectively.

Fig. 9(b) shows 256-core results. TCMalloc suffers spurious conflicts among tasks that access the same allocator metadata, and is gmean 25× slower than the ideal allocator. By contrast, capalloc is only gmean 30% slower than the ideal allocator. These overheads are in line with those in the single-core system, demonstrating capalloc's scalability. capalloc is gmean 20× faster than TCMalloc—from 3× (nocsim) to 69× (silo).

D. Capsules case study: Disk-backed key-value store

The previous case study showed that Capsules avoid needless conflicts; we now show the benefits of letting speculative tasks perform controlled parallel I/O. We implement a simple disk-backed key-value store that runs the YCSB [19] benchmark with 4 KB tuples and 80/20% read/write queries. Each query runs within a single speculative task. The key-value store keeps only some of the tuples in main memory. If the requested tuple is not in main memory, it must be fetched from disk and another tuple must be evicted, writing it back to disk if it is dirty.

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



(a) Disk bandwidth usage (b) Performance at 2% miss rate

Fig. 10. Disk utilization of spec and capsule variants of a key-value store.

We evaluate both strategies on a 256-core system with an NVMe SSD.[3] Fig. 10(a) shows how disk bandwidth grows with miss rate (which we control by varying the memory footprint). `spec` tops out at 63 MB/s, far below the disk's bandwidth, due to its serialized I/O. By contrast, `capsule` fully saturates disk bandwidth, achieving 1671 MB/s. Fig. 10(b) shows that, with a 2% miss rate (where both variants are I/O-bound), `capsule` achieves 24× the throughput of `spec`. These results show that concurrent system calls can be highly beneficial, and Capsules successfully unlock this benefit for speculative tasks.

VII. Additional Related Work

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

A. Task scheduling and synchronization

Prior work has investigated hardware support for scheduling and synchronization of *either* speculative or non-speculative tasks. On the speculative side, prior techniques enable threads to speculate past barriers [36, 58, 82], and avoid aborts on known dependences by stalling [96] or pipelining [91]. On the non-speculative side, prior work has proposed hardware-accelerated task-stealing [53, 81] and dataflow [15, 28, 33, 66] schedulers. The lack of shared synchronization mechanism hinders HTM, where mixing transactional and conventional synchronization is unsafe [26, 92]. Prior work has crafted software primitives that bypass transactional mechanisms [26, 92] or toggle between transactional and lock-based synchronization [78].

By contrast, Espresso's timestamps and locales facilitate coordination *across* speculative and non-speculative tasks. This opens the door to `MAYSPEC`, which allows the system to dynamically choose to execute tasks speculatively or non-speculatively. Moreover, timestamps and locales offer more performance for non-speculative tasks than shared-memory barriers and locks. Timestamps are essentially hardware-accelerated barriers [7, 50, 83]. Locales are handled by the task dispatch logic, so they are more efficient than hardware-accelerated locks [49, 56, 99], as they eliminate spinning within a task. Locales also enable locality-aware task mapping.

B. Restricted vs. unrestricted speculative tasks

TLS systems are unrestricted: their tasks can run arbitrary code, although only the earliest active task may run a system call or exception handler. Most HTMs are restricted: they forbid transactions from invoking irrevocable actions, which hinders programmability. OneTM [9] and TCC [36] permit unrestricted transactions. Our promotion technique lies between OneTM-serialized, which pauses all other threads, and OneTM-concurrent, which keeps all other threads running but requires in-memory metadata to support unbounded read/write sets. By contrast, Espresso keeps only speculative tasks running through a promotion. TCC, like TLS, does not support non-speculative parallelism (all code runs speculatively except the transaction with commit permission).

[3] We model a Samsung 960 PRO, which supports 440K/360K IOPS for random 4 KB reads/writes, with minimum latencies of 70/20 μs [80].

TABLE IV
Comparison of prior systems and Espresso.

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

[a] Most HTMs are unordered (Sec. VII-D).
[b] Most HTMs are restricted (Sec. VII-B).

C. Open-nested transactions

Some speculative tasks must perform operations that would be expensive or incompatible with their hardware speculation mechanisms. Escape actions (Sec. II-C) are one prior solution for HTMs, as are open-nested transactions [59, 63, 64], which run within another transaction and commit immediately after finishing, before its enclosing transaction commits. Like Capsules, open-nested transactions still use ordinary conflict detection to preserve atomicity when accessing data shared by other transactions. Like escape actions and Capsules, open-nested transactions use abort handlers to undo their effects. Unfortunately, open-nested transactions are also unsafe with speculative forwarding because open-nested transactions may lose data and control-flow integrity and then perform harmful writes and commit.

D. Transactional memory and order

Some hardware [16, 36, 71] and software [12, 32] TMs let programmers control the commit order among transactions, bridging the gap between TM and TLS. Other TMs order transactions internally, either to avoid pathologies [11, 62] or to implement conflict serializability [4, 29, 44, 72, 74]. However, this order is not controllable by programmers.

VIII. Conclusion

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

Acknowledgments

We sincerely thank Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Guowei Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants CAREER-1452994 and SHF-1814969, NSF/SRC grant E2CDA-1640012, and C-FAR, one of six SRC STARnet centers by MARCO and DARPA. Mark C. Jeffrey was supported by a Facebook Fellowship and Hyun Ryong Lee was supported in part by a Kwanjeong Educational Foundation scholarship.

REFERENCES

[1] "9th DIMACS Implementation Challenge: Shortest Paths," http://www.dis.uniroma1.it/~challenge9, archived at https://perma.cc/5KYT-YM36, 2006.

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

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

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

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

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

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

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

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

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

[11] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Proc. ISCA-34*, 2007.

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

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

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

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

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

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

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

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

[20] L. Dalessandro and M. L. Scott, "Strong isolation is a weak idea," in *TRANSACT*, 2009.

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

[22] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, 2011.

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

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

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

[26] P. Dudnik and M. M. Swift, "Condition variables and transactional memory: Problem or opportunity?" in *TRANSACT*, 2009.

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

[28] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task Superscalar: An out-of-order task pipeline," in *Proc. MICRO-43*, 2010.

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

[30] 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.

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

[32] M. A. Gonzalez-Mesa, E. Gutierrez, E. L. Zapata, and O. Plata, "Effective transactional memory execution management for improved concurrency," *ACM TACO*, vol. 11, no. 3, 2014.

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

[34] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proc. PPoPP*, 2008.

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

[36] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proc. ISCA-31*, 2004.

[37] F. M. Harper and J. A. Konstan, "The MovieLens datasets: History and context," *ACM TiiS*, vol. 5, no. 4, 2015.

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

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

[40] 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.

[41] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. ISCA-20*, 1993.

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

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

[44] 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.

[45] D. R. Jefferson, "Virtual time," *ACM TOPLAS*, vol. 7, no. 3, 1985.

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

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

[48] 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.

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

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

[51] 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.

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

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

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

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

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

[57] 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.

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

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

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

[61] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. IISWC*, 2008.

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

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

[64] J. E. B. Moss, "Open nested transactions: Semantics and support," in *Workshop on Memory Performance Issues*, 2006.

[65] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase Reconciliation for Contended In-Memory Transactions." in *Proc. OSDI-11*, 2014.

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

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

[68] 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.

[69] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proc. SPAA*, 2011.

[70] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proc. PLDI*, 2011.

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

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

[73] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proc. ASPLOS-X*, 2002.

[74] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Proc. MICRO-41*, 2008.

[75] 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.

[76] 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.

[77] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis technique for parallelizing compilers," *ACM TOPLAS*, vol. 19, no. 6, 1997.

[78] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel, "TxLinux: Using and managing hardware transactional memory in an operating system," in *Proc. SOSP-21*, 2007.

[79] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proc. PPoPP*, 2010.

[80] Samsung, "Samsung SSD 960 PRO M.2 Data Sheet," 2017.

[81] D. Sanchez, R. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *Proc. ASPLOS-XV*, 2010.

[82] T. Sato, K. Ohno, and H. Nakashima, "A mechanism for speculative memory accesses following synchronizing operations," in *Proc. IPDPS*, 2000.

[83] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proc. ASPLOS-VII*, 1996.

[84] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proc. PPoPP*, 2013.

[85] 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.

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

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

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

[89] 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.

[90] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. SOSP-24*, 2013.

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

[92] H. Volos, N. Goyal, and M. M. Swift, "Pathological interaction of locks with transactional memory," in *TRANSACT*, 2008.

[93] 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.

[94] 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.

[95] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing," in *Proc. SC13*, 2013.

[96] 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.

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

[98] Y. Zhang, V. Kiriansky, C. Mendis, S. P. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *Proc. IEEE BigData*, 2017.

[99] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, "Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures," in *Proc. ISCA-34*, 2007.

[100] C. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," in *TRANSACT*, 2006.