Unlocking Ordered Parallelism with the Swarm Architecture

Swarm is a parallel architecture that exploits ordered parallelism. It executes tasks speculatively and out of order and can scale to large core counts and speculation windows. The authors evaluate Swarm on graph analytics, simulation, and database benchmarks. At 64 cores, Swarm outperforms sequential implementations of these algorithms by 43 to 117 times and state-of-the-art software-only parallel algorithms by 3 to 18 times.

Mark C. Jeffrey
Suvinay Subramanian
Massachusetts Institute of Technology

Cong Yan
University of Washington

Joel Emer
Nvidia

Daniel Sanchez
Massachusetts Institute of Technology

Multicores are now pervasive, but they still provide limited architectural support for parallelization, constraining the range of applications that can exploit them. Thus, it is crucial to explore new architectural mechanisms to efficiently exploit as many types of parallelism as possible. Doing so makes parallel systems more versatile and easier to program and, for many applications, it is the only way to improve performance.

Fundamentally, parallelizing a program comprises two main steps: dividing work into tasks and enforcing synchronization among tasks with potential data dependences to ensure correct behavior. Tasks might be created dynamically at runtime. Broadly, we can distinguish two classes of parallelism, unordered and ordered, that place different demands on the system.

In unordered parallel programs, available tasks can execute and complete in any order. Tasks can have data dependences that are not known a priori. In this case, the programmer must use some form of explicit synchronization, such as locks or transactions, to arbitrate accesses to shared data. Unordered parallelism incurs small overheads in current multicores as long as tasks synchronize infrequently and are large enough to amortize task-management costs in software (such as scheduling and load balancing).

By contrast, ordered parallel programs consist of tasks that must follow a total or partial order. Tasks can have data dependences that are unknown a priori, but synchronization is implicit, determined by their order constraints. When tasks create new children tasks, they schedule them to run at a specific future time. The combination of order constraints and unknown data dependences makes ordered parallelism hard to exploit in current multicores,[1] because runtime overheads negate parallelism's benefits.

The goal of this work is to design efficient architectural support for ordered parallelism. This brings two main benefits. First, many key algorithms have plentiful ordered parallelism but little unordered parallelism, so they scale poorly. Second, although ordered parallelism is more demanding on the system, it is

0272-1732/16/$33.00 © 2016 IEEE Published by the IEEE Computer Society

simpler and more general than unordered parallelism. Thus, efficient support for order also eases parallel programming, because applications with unordered parallelism often have simpler ordered implementations.

In our paper presented at the 48th International Symposium on Microarchitecture,[2] we introduced Swarm, an architecture that exploits ordered parallelism efficiently. Swarm relies on a co-designed execution model and microarchitecture to scale. Specifically, we contribute four novel techniques:

- an execution model based on tasks with programmer-specified time stamps that conveys order constraints to hardware;
- a hardware task-management scheme that features speculative task creation and dispatch, drastically reducing task management overheads, and implements a large speculation window;
- a scalable conflict-detection scheme that leverages eager versioning to, upon mispeculation, selectively abort the mispeculated task and its dependents; and
- a distributed commit protocol that allows ordered commits without serialization, supporting multiple commits per cycle with modest communication.

We evaluated Swarm on six graph analytics, simulation, and database benchmarks. At 64 cores, Swarm outperformed sequential implementations of these algorithms by 43 to 117 times, and it outperformed state-of-the-art parallel implementations on conventional multicores by 2.7 to 18 times. Besides achieving near-linear scalability on algorithms that are often considered sequential, the resulting Swarm programs are almost as simple as their sequential counterparts, because they do not use explicit synchronization.

Understanding Ordered Parallelism

Applications with ordered parallelism have three key features.[3] First, they comprise tasks that must execute in some total or partial order. Second, tasks are not known in advance. Instead, tasks dynamically create children tasks and schedule them to run at a future time, resulting in different task creation and execution orders. Third, tasks can have data dependences that are not known a priori.

Example: Dijkstra's Algorithm

To illustrate the challenges in parallelizing these applications, consider Dijkstra's single-source shortest paths (`sssp`) algorithm.[4] `sssp` finds the shortest distance between some source node and all other nodes in a graph with weighted edges. Figure 1a shows the sequential code for `sssp`, which uses a priority queue to store tasks. Each task operates on a single node and is ordered by its projected distance to the source node. `sssp` relies on task order to guarantee that the first task to visit each node comes from a shortest path. This task sets the node's distance and enqueues tasks to visit each neighbor. Later tasks visiting the same node do nothing.

Figure 1b shows an example graph, and Figure 1c shows the tasks that `sssp` executes to process this graph. Figure 1c also shows the execution order of each task (projected distance to source node) in the *x*-axis and outlines both parent-child relationships and data dependences. For example, task A at distance 0, denoted $(A, 0)$, creates children tasks $(C, 2)$ and $(B, 3)$; task $(B, 3)$ writes to node B and $(B, 4)$ reads it, so they have a data dependence.

A distinctive feature of many programs with ordered parallelism is that task creation and execution orders are different: children tasks are not immediately runnable, but are subject to a global order influenced by all other tasks in the program. For example, in Figure 1c, $(C, 2)$ creates $(B, 4)$, but running $(B, 4)$ immediately would produce the wrong result, because $(B, 3)$, created by a different parent, must run first. Sequential implementations of these programs use scheduling data structures, such as priority queues or first-in, first-out queues, to process tasks in the right order. This is a key reason why thread-level speculation (TLS),[5–8] which speculatively parallelizes sequential programs, cannot exploit ordered parallelism: the scheduling data structures introduce false data dependences among otherwise independent tasks. For further information, see the "Related Work in Thread-Level Speculation" sidebar.

Given these order constraints, where is the parallelism? The key insight is that independent tasks (for example, those visiting different

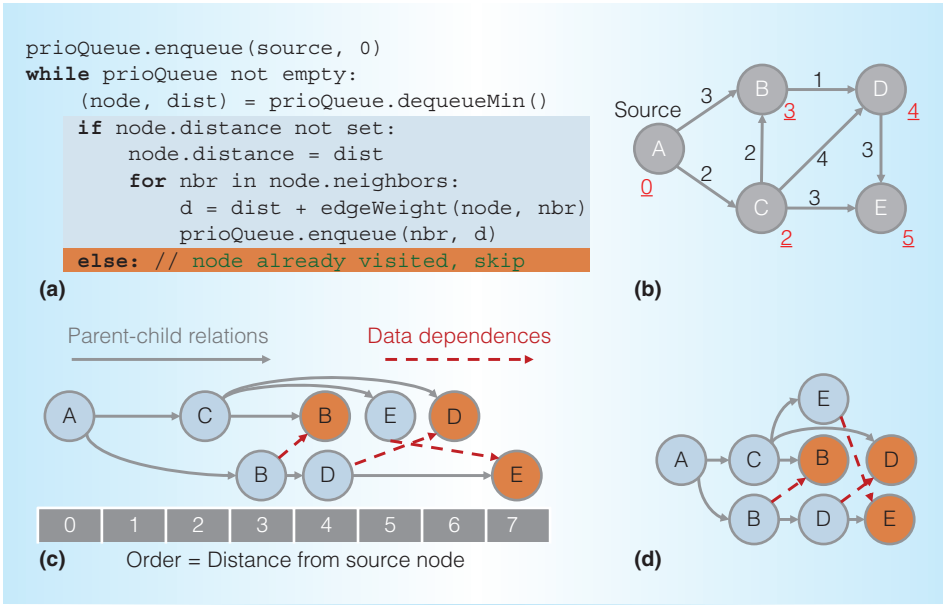

Figure 1. Dijkstra's single-source shortest paths (`sssp`) algorithm has plentiful ordered parallelism. (a) Dijkstra's `sssp` code, highlighting the unvisited and visited paths that each task can follow. (b) Example graph and resulting shortest-path distances (underlined). (c) Tasks executed by `sssp`. Each task shows the node it visits. Tasks that visit the same node have a data dependence. (d) A correct speculative schedule that achieves twice the parallelism.

nodes in `sssp`) can execute concurrently and out of order without violating correctness. For example, Figure 1d shows a correct parallel schedule for `sssp` tasks. Tasks in the same *x*-axis position execute concurrently. This schedule achieves twice the parallelism of a serial schedule in this small graph, and larger graphs permit even more parallelism. The parallel schedule produces the correct result because, although it elides order constraints among independent tasks, it respects data dependences.

Unfortunately, tasks and their data dependences are not known in advance. Therefore, to elide unnecessary order constraints, we must resort to speculative execution. Specifically, for every task other than the earliest active task, Swarm speculates that there are no earlier data-dependent tasks, and it executes the task anyway. If this guess is wrong, Swarm detects dependence violations and aborts offending tasks to preserve correctness.

Opportunities and Challenges

To motivate and guide Swarm's design, we characterized six applications, ranging from graph analytics to simulation and databases. Our full paper contains a detailed limit study, from which we gleaned three key insights: speculative parallelism is plentiful, tasks are tiny, and a large speculative window is needed.[2]

Speculative parallelism is plentiful. Just as Figure 1d is a valid schedule with twice the parallelism for `sssp` on a small graph, we determine the shortest possible schedule that respects data dependences for each benchmark when using large, representative inputs. All applications exhibit more than 100 times maximum parallelism. For example, `sssp` reached 793 times on a large road graph. Thus, most order constraints are superfluous, making speculative execution attractive.

Tasks are tiny. Across the benchmark suite, tasks are very short, ranging from a few tens of instructions (`sssp`) to a few thousand. Tasks also have small read and write sets. For example, `sssp` tasks read 5.8 64-bit words and write 0.4 words on average. Task-scheduling overheads in software overwhelm the

Related Work in Thread-Level Speculation

Prior work has investigated thread-level speculation (TLS) schemes to parallelize sequential programs.[1–5] TLS schemes ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Although TLS schemes can elide order constraints, we find that two key problems prevent them from exploiting ordered parallelism: their execution model limits application parallelism and prior implementations suffer from scalability bottlenecks.

The TLS Execution Model Limits Parallelism

To run under TLS, ordered algorithms must be expressed as sequential programs, but sequential implementations often limit parallelism. Consider the code in Figure 1a in the main article, in which each iteration dequeues a task from the priority queue and runs it, potentially enqueuing more tasks. Data dependences in the priority queue, not among tasks themselves, cause frequent conflicts and aborts. For example, iterations that enqueue high-priority tasks often abort all future iterations.

Through a limit study, we have shown that TLS parallelism is meager in most of our benchmarks, even with idealizations such as perfect speculation, an infinite task window, and no communication delays.[6] For example, because of frequent dependences on the priority queue, the sequential `sssp` implementation has only 1.1 times parallelism.

The root problem is that loops and method calls, the control-flow constructs of TLS schemes, are insufficient to express the order constraints among these tasks. By contrast, Swarm implements a more general execution model with time-stamp-ordered tasks to avoid software queues, and implements hardware priority queues integrated with speculation mechanisms, avoiding spurious aborts due to queue-related references.

TLS Scalability Bottlenecks

Although prior work has developed scalable versioning and conflict-detection schemes, two challenges limit TLS performance with large speculation windows and small tasks: unselective aborts and limited commit throughput.

Forwarding Versus Selective Aborts

Most TLS schemes find it desirable to forward data written by an earlier, still-speculative task to later reader tasks. This prevents later tasks from reading stale data, reducing mispeculations on tight data dependences. However, it creates complex chains of dependences among speculative tasks. Thus, upon detecting mispeculation, most TLS schemes abort the task that caused the violation and all later speculative tasks.[1–3,5,7]

We similarly find that forwarding speculative data is crucial for Swarm. However, although it is reasonable to abort all later tasks with small speculative windows (2 to 16 tasks are typical in prior work), Swarm has a 1,024-task window, which makes unselective aborts unreasonable. To address this, our novel conflict-detection scheme forwards speculative data and selectively aborts only dependent tasks upon mispeculation.

Commit Serialization

Prior TLS schemes enforce in-order commits by passing a token among ready-to-commit tasks.[1–3,7] Each task can commit only when it has the token, and passes the token to its immediate successor when it finishes committing. This approach cannot scale to the commit throughput that Swarm needs. For example, with 64-cycle tasks, a 64-core system should commit one task per cycle on average. Even if commits were instantaneous, the token-passing latency makes this throughput unachievable.

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

References

1. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecutre*, 1995, pp. 414–425.
2. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. 8th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 58–69.
3. J. Renau et al., "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," *Proc. 19th Ann. Int'l Conf. Supercomputing*, 2005, pp. 179–188.
4. J.G. Steffan and T. Mowry, "The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization," *Proc. 4th Int'l Symp. High-Performance Computer Architecture*, 1998, pp. 2–13.
5. M.J. Garzarán et al., "Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors," *Proc. 9th Int'l Symp. High-Performance Computer Architecture*, 2003, pp. 191–202.
6. M.C. Jeffrey et al., "A Scalable Architecture for Ordered Parallelism," *Proc. 48th Int'l Symp. Microarchitecture*, 2015, pp. 228–241.
7. J.G. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, 2000, pp. 1–12.

useful work of short tasks.[1] Moreover, order constraints prevent runtimes from grouping tasks into coarser-grained units to amortize overheads. Hardware support for task management is needed to reduce these overheads.

A large speculative window is needed. To find how far ahead of the earliest active task Swarm needs to execute, we studied how the maximum parallelism changes with a limited task window. With a T-task window, when finding the shortest schedule, we do not schedule an independent task until all work more than T tasks behind has finished. In five out of the six applications, small windows severely limited parallelism. For example, parallelism in `sssp` dropped from 793 times with an infinite window to 178 times with a 1,024-task window to 26 times with a 64-task window. Therefore, the architecture must support many more speculative tasks than cores to uncover enough parallelism.

Swarm Execution Model

Swarm programs consist of time-stamped tasks. Each task can access arbitrary data and create children tasks with any time stamp greater than or equal to its own. Swarm guarantees that tasks appear to execute in time-stamp order. If multiple tasks have the same time stamp, Swarm chooses an order among them.

Using time stamps decouples task creation and execution orders: software conveys new work to hardware as soon as it is discovered rather than in the order it needs to run, exposing a large amount of parallelism.

Programs leverage the Swarm execution model through a simple API. Figure 2 shows the Swarm implementation of `sssp` and illustrates the execution model and API. The code closely resembles the sequential implementation from Figure 1a—there is no explicit synchronization or thread management.

In our API, each task executes a function that takes a time stamp and an arbitrary number of additional arguments. Figure 2 defines a single task function, `ssspTask`. Because `sssp` tasks are ordered by their node's projected distance to the source, that distance is used directly as a task time stamp.

```
void ssspTask(Timestamp dist, Node& n) {
  if (n.distance == UNSET) {
    n.distance = dist;
    for (Node& m: n.neighbors) {
      Timestamp mDist = dist + edgeWeight(n, m);
      swarm::enqueue(&ssspTask, mDist, m);
    }
  } else {} // node n already visited, skip
}
swarm::enqueue(&ssspTask, 0, sourceNode);
swarm::run();
```

Figure 2. Swarm implementation of Dijkstra's `sssp` algorithm. The code is similar to the sequential implementation (see Figure 1a). Parallelism and synchronization are implicit.

In our API, tasks can create children tasks by calling `swarm::enqueue` with the appropriate task function, time stamp, and arguments. For example, if its own node is unvisited, `ssspTask` creates one child task for each neighbor, with the neighbor's projected distance as the time stamp. Because tasks appear to execute in time-stamp order, the first task to visit each node will come from a shortest path.

Finally, a program invokes Swarm by enqueuing some initial tasks with `swarm::enqueue` and calling `swarm::run`, which returns control when all tasks finish. Figure 2 enqueues one initial task that visits the source node before initiating speculative execution.

Swarm Microarchitecture

The Swarm microarchitecture exploits ordered parallelism by executing tasks speculatively and out of order. Swarm introduces modest changes to a tiled, cache-coherent multicore, shown in Figure 3. 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. Every tile is augmented with a task unit that queues, dispatches, and commits tasks.

Swarm is carefully designed to support tiny tasks and a large speculation window efficiently. Four key ingredients make this possible: low-overhead hardware task management, large task queues, scalable speculation mechanisms, and high-throughput ordered commits.

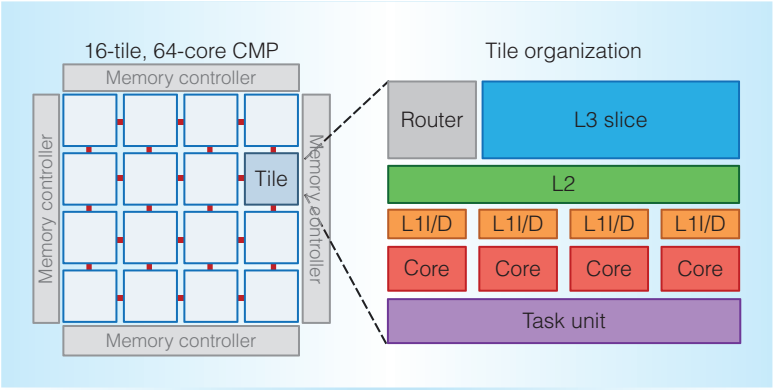

Figure 3. Swarm chip multiprocessor and tile configuration. Swarm augments each tile with a task unit that queues tasks, dispatches them to local cores, and determines when they commit.

Hardware Task Management

Each tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Task units communicate only to send new tasks to each other to maintain load balance and, infrequently, to determine which finished tasks can be committed. Swarm executes every task except the earliest active task speculatively. To uncover enough parallelism, task units can dispatch any available task to cores, no matter how distant in program order. Tasks also can run even if their parent is still speculative.

Each task is represented by a task descriptor that contains its function pointer, 64-bit time stamp, and arguments. Cores interact with task units using enqueue and dequeue instructions. Cores enqueue each task descriptor to a randomly chosen tile and dequeue tasks for execution from the local task unit. Successful dequeues initiate speculative execution at the task's function pointer and make the task's time stamp and arguments available in registers. Dequeues block until a task is available or all tasks have committed.

Task Unit Queues

The task unit has two main structures: a task queue that holds task descriptors for every task in the tile, and 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. Figure 4 shows how these queues are used throughout a task's lifetime. Each new task allocates a task queue entry and holds it until commit time. Tasks do not necessarily arrive in priority order. For example, Figure 4a shows an arriving task with time stamp 7, while tasks with time stamps 8 and higher had previously been enqueued. Tasks are dispatched in time-stamp order: the task queue fulfills each dequeue request by dispatching its lowest-time-stamp idle task. For example, in Figure 4b, the task queue dispatches the task with time stamp 7. Each task allocates a commit queue entry when it finishes execution, as Figure 4c shows. Commit queue entries are deallocated when the task commits or aborts.

Both queues allocate entries independently of task priority order, as Figure 4 shows; they manage their free space with free lists. To find the highest-priority idle task in the task unit, Swarm uses ternary content-addressable memories.[9] (See our full paper for details.[2])

Because programs can enqueue tasks with arbitrary time stamps, task and commit queues can fill up. This requires some simple actions to ensure correct behavior. Specifically, idle tasks whose parents are nonspeculative can be spilled to memory to free task queue entries. For all other tasks, queue resource exhaustion is handled either by stalling the enqueuer or aborting higher-time-stamp tasks to free space. Again, our full paper contains more details.[2]

Ordered Speculation

The key requirements for speculative execution in Swarm are fast commits and a large speculation window. To this end, we adopt eager versioning, which stores speculative data in place and logs old values. Eager versioning makes commits fast and requires a minimal amount of state per task, but aborts are slow. However, Swarm's execution model makes conflicts rare, so eager versioning is the right tradeoff.

Swarm's speculative execution borrows from Log-based Transitional Memory (LogTM).[10] The per-task speculative state includes read-set and write-set Bloom-filter signatures, an undo log pointer, and child pointers. Because speculation happens at the task level, there are no register checkpoints, unlike in Hardware TM and TLS. Each core and commit queue entry holds this state. Swarm offers the following key

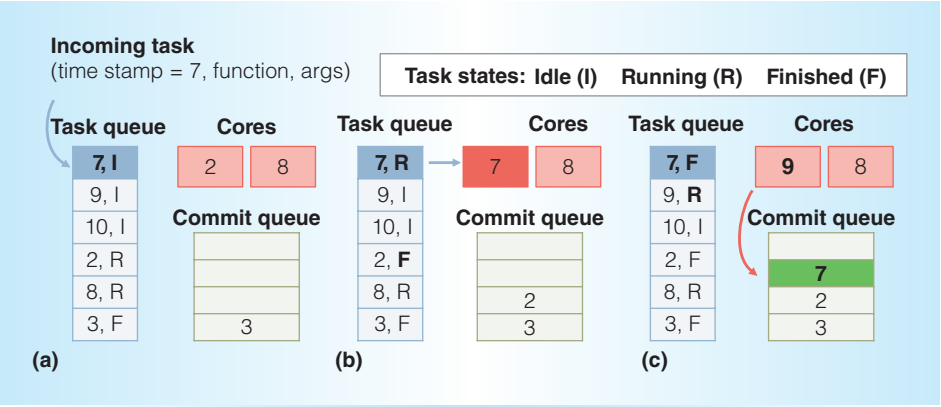

Figure 4. Task queue, core, and commit queue utilization throughout a task's lifetime. (a) Task with time stamp 7 arrives and is queued. (b) Task with time stamp 2 finishes and task with time stamp 7 starts running. (c) Task with time stamp 7 finishes and task with time stamp 9 starts running.

contributions over these and other speculation schemes that enable it to scale to large speculative windows: conflict detection that forwards data from still-speculative tasks, new techniques to reduce the cost and frequency of conflict checks, and conflict resolution that selectively aborts only a violating task's dependents.

As in LogTM-SE, as a task executes, hardware automatically performs conflict detection on every read and write. Then it inserts the read and written addresses into the Bloom filters, and, for every write, it saves the old memory value in a memory-resident undo log.

Enforcing a total order. The execution model permits tasks with the same programmer-assigned time stamp. To enforce atomicity, the hardware chooses a total order among same-time-stamp tasks. Task units assign each dispatched task a unique virtual time, the concatenation of the task's time stamp and a unique tiebreaker. Tasks retain their virtual time until they commit. Tiebreakers ensure that even if a child task has the same time stamp as its parent, the child's virtual time is higher.

Conflict detection with forwarding. Conflicts can arise when a task uses a line that was previously used by a later-virtual-time task. Conflicts are detected at cache-line granularity. Suppose two tasks, t_1 and t_2, are running or finished, and t_2 has a later virtual time. A read of t_1 to a line written by t_2 or a write to a line read or written by t_2 causes t_2 to abort. However, t_2 can access data written by t_1 even if t_1 is still speculative. Thanks to eager versioning, t_2 automatically uses the latest copy of the data—there is no need for speculative data-forwarding logic.[11]

Memory requests for conflict-checked addresses contain the requester's virtual time. When a tile receives an invalidation request, it checks for the address in all local read sets and write sets and detects any order-violating conflicts. To make conflict detection efficient, we must reduce the number of tile-wide conflict checks and the cost of each tile-wide conflict check.

Using caches to filter conflict checks. Swarm exploits the cache hierarchy to filter conflict checks at each level by using a key invariant: when a task with virtual time T installs a line in the (L1 or L2) cache, that line has no conflicts with tasks of virtual time greater than T. As long as the line stays cached with the right coherence permissions, it stays conflict-free. Because conflicts arise only when tasks access lines out of virtual time order, if another task with virtual time $U > T$ accesses the line, it is also guaranteed to have no conflicts. However, an access from a task with virtual time $U < T$ must trigger a conflict check in the next cache level. Swarm maintains the invariant above by modifying each cache level. First, when a core dequeues a task with a lower virtual time than the one it just finished, the L1 flushes lines that were previously accessed speculatively. Second,

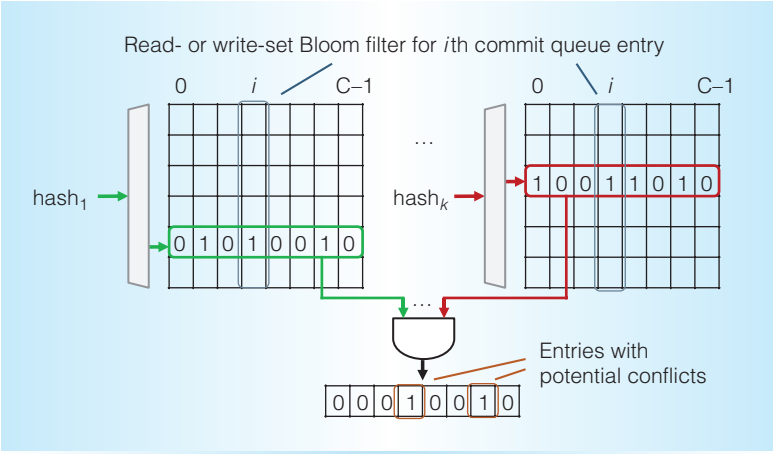

Figure 5. Commit queues store read- and write-set Bloom filters by columns, so a single access reads one bit from all entries. All entries are checked in parallel.

each L2 set has an associated canary virtual time. This canary stores the lowest virtual time an accessing task can have that does not require a global conflict check. Finally, exactly as in LogTM,[10] the L3 directory uses memory-backed sticky bits to check only those tiles whose tasks might have accessed the line.

Our full paper shows that this invariant is safe and further illustrates hierarchical conflict checks.[2]

Efficient commit-queue conflict checks. On every L2 access, and on some global conflict checks, all speculative tasks in a tile's commit queue are checked. To enable large commit queues (for example, 64 tasks per queue), these commit queue checks must be efficient. To this end, we leverage the fact that querying a K-way Bloom filter requires reading only one bit from each way, as shown in Figure 5. Bloom filter ways are stored in columns, so a single 64-bit (row) access per way reads all the necessary bits. Reading and ANDing all ways yields a word that indicates potential conflicts. For each queue entry whose position in this word is set, its virtual time is checked; those with virtual time higher than the issuing task's must be aborted.

Selective aborts. Prior TLS schemes that forward speculative data perform unselective aborts, aborting the task that caused the violation and all later speculative tasks. This would waste too much work in Swarm. Instead, Swarm uses a novel conflict-resolution scheme that performs selective aborts, aborting only dependents of the order-violating task: its children and other tasks that have accessed data in its write set. This technique reuses the conflict-detection machinery to discover and abort dependent tasks as needed. This avoids explicitly tracking intertask dependences, which would require space that grows quadratically with the number of speculative tasks.

Hardware aborts a task t in three steps:

1. Notify t's children to abort and be removed from their task queues.
2. Walk t's undo log in last-in, first-out order, restoring old values. If one of these writes conflicts with a later-virtual-time task, wait for it to abort and continue t's rollback.
3. Clear t's read and write sets and free its commit queue entry.

Applied recursively, this procedure selectively aborts all dependent tasks, as shown in Figure 6. Undo-log writes (for example, A's second "wr 0x10" in Figure 6) are normal conflict-checked writes, issued with the task's time stamp to detect all later readers and writers.

High-Throughput Ordered Commits

A key requirement for scalable speculative execution with tiny tasks is enabling high-throughput ordered commits. For example, with 64-cycle tasks, a 64-core system should commit one task per cycle on average. Prior techniques from TLS rely on serial token-passing and are far below this throughput.

Swarm adapts the virtual-time algorithm,[12] common in parallel discrete event simulation,[13] for this purpose. Figure 7 illustrates the protocol: tiles communicate to discover which is the earliest active task in the system. They periodically send the smallest virtual time of any unfinished (idle or running) task to an arbiter. The arbiter computes the minimum virtual time of all unfinished tasks, called the *global virtual time* (GVT), and broadcasts it to all tiles. Any task with virtual time less than the GVT commits: it precedes the earliest active task, so it cannot have violated the task total priority order.

The key insight is that, by combining the virtual time algorithm with Swarm's large

commit queues, commit costs are amortized over many tasks. A single arbiter update often causes many finished tasks to commit. This means GVT updates can be done sparingly (for example, every 200 cycles), requiring minimal bandwidth. Finally, eager versioning makes commits fast: a task is committed by freeing its task and commit queue entries, which is a single-cycle operation. Thus, when many tasks commit at once, queue space becomes available quickly.

Putting It All Together

Figure 8 summarizes Swarm's hardware changes. Swarm adds task units and a GVT arbiter and modifies cores and caches. For a 16-tile, 64-core system with 2,048-bit Bloom filters, Swarm's structures consume 0.55 mm^2 per four-core tile, or 8.8 mm^2 per chip, a modest cost. See our full paper for detailed cost breakdowns.[2]

In summary, Swarm's costs are moderate, and, in return, confer significant speedups.

Swarm Evaluation

We use six challenging workloads to evaluate Swarm:

- `bfs` finds the breadth-first tree of an arbitrary graph.
- `sssp` is Dijkstra's algorithm (see Figure 2).
- `astar` uses the A* pathfinding algorithm to find the shortest route between two points in a road map.
- `msf` is Kruskal's minimum-spanning forest algorithm.
- `des` is a discrete-event simulator for digital circuits. Each task represents a signal toggle at a gate input.
- `silo` is an in-memory online transaction processing database.[14]

For most benchmarks, we use tuned serial and state-of-the-art parallel versions from existing suites (detailed in our full paper[2]). We then port each serial implementation to Swarm. Swarm versions use fine-grained tasks, but they use the same data structures and perform the same work as the serial version, so differences between serial and Swarm versions stem from parallelism, not other optimizations.

We wrote our own serial and Swarm `astar` implementations. `astar` is notoriously difficult to parallelize—to scale, prior work in parallel pathfinding sacrifices solution quality for speed.[15] Thus, we do not have a software-only parallel implementation.

We port `silo` to show that Swarm can extract ordered parallelism from applications that are typically considered unordered.

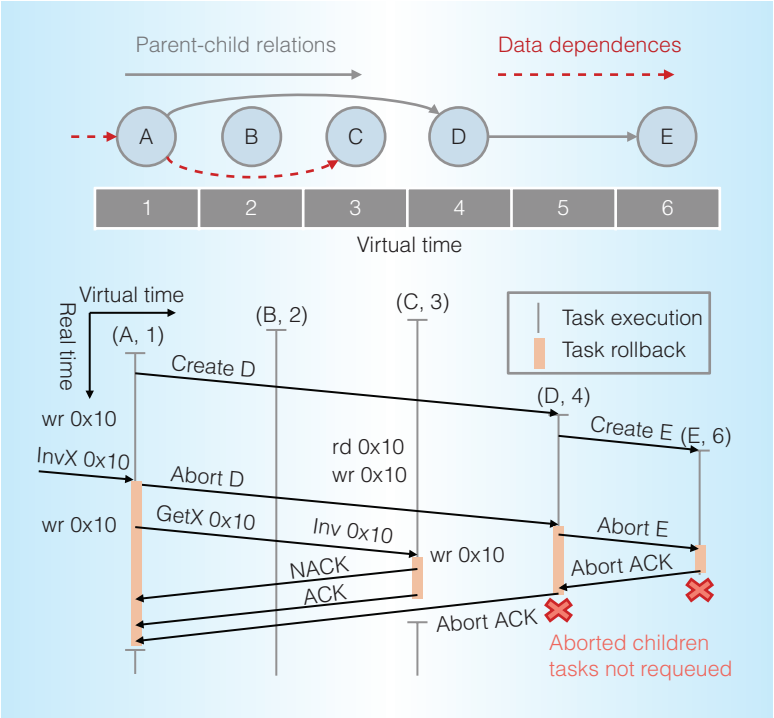

Figure 6. Selective abort protocol. Suppose (*A*, 1) must abort after it writes 0x10. (*A*, 1)'s abort squashes child (*D*, 4) and grandchild (*E*, 6). During rollback, *A*'s corrective write also aborts (*C*, 3), which read *A*'s speculative write to 0x10. (*B*, 2) is independent and thus not aborted.

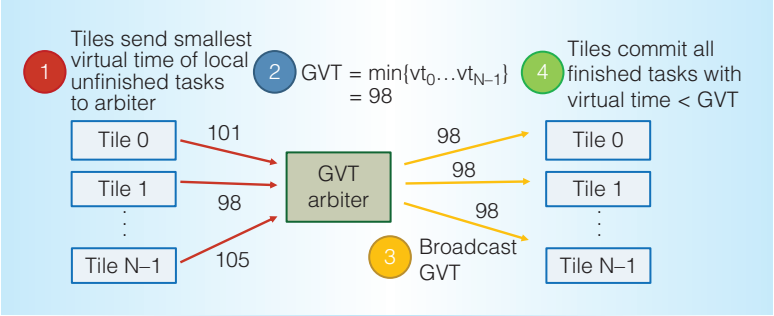

Figure 7. Scalable ordered commits protocol. Tiles periodically communicate with an arbiter to determine the earliest active task in the system. All tasks that precede this earliest active task can safely commit.

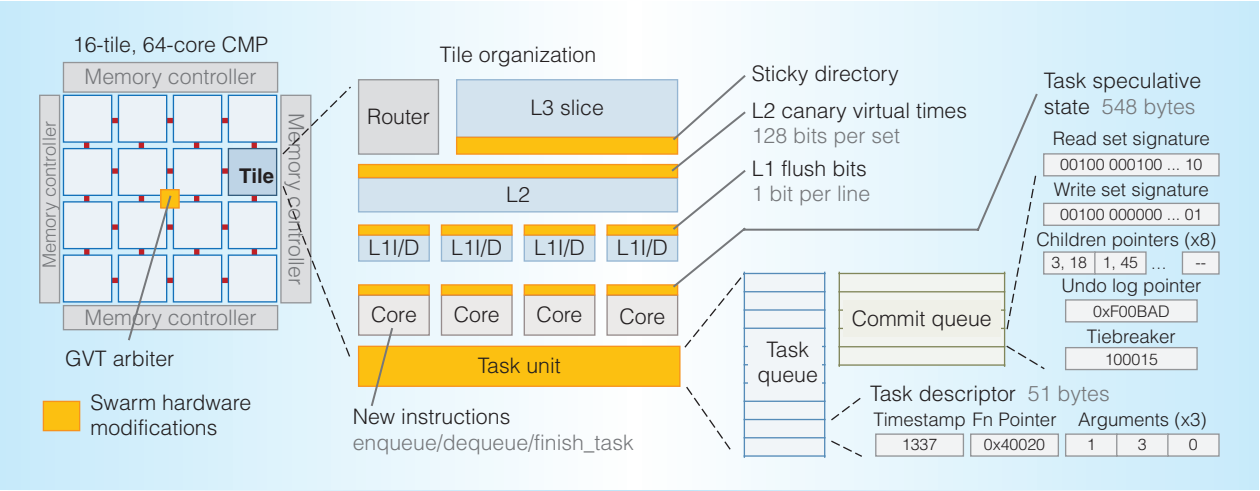

Figure 8. Summary of Swarm hardware modifications. Swarm augments a tiled chip multiprocessor with a GVT arbiter, a task unit on each tile, and modifications to cores and caches. These enhancements require moderate cost.

Database transactions are unordered in `silo`. We decompose each transaction into many small ordered tasks to exploit intratransaction parallelism. Tasks from different transactions use disjoint time-stamp ranges to preserve atomicity. This exposes significant fine-grained parallelism within and across transactions.

We choose representative inputs for each algorithm.[2] All benchmarks have serial runtimes of more than two billion cycles. Note that some inputs can offer plentiful trivial parallelism to a software algorithm. Specifically, on large, shallow graphs (for example, 10 million nodes and 10 levels), a simple bulk-synchronous `bfs` that operates on one level at a time scales well.[16] But we use a graph with 7.1 million nodes and 2,799 levels, so `bfs` must speculate across levels to uncover enough parallelism. Our full paper reports `silo` results when running the TPC-C benchmark with 1 to 64 warehouses. Here, we show the one-warehouse results only.

We simulate systems with up to 64 cores. The 64-core chip has 16 tiles, as shown in Figure 8. Each core has 16-Kbyte private L1 caches, each tile has a 256-Kbyte L2 cache, and all tiles share a 16-Mbyte L3 cache distributed across tiles. Swarm uses 64 task queue entries per core (4,096 total), 16 commit queue entries per core (1,024 total), and 2,048-bit, eight-way Bloom filters for conflict checks. Our full paper contains the complete description of our methodology.[2]

Swarm versus Software Implementations

Figure 9 compares the performance of the Swarm and software-only versions of each benchmark. Each graph shows the speedup of the Swarm and software-parallel versions over the tuned serial version running on a system of the same size, from 1 to 64 cores. In this experiment, per-core queue and L2 and L3 capacities are kept constant as the system grows, so systems with more cores have higher queue and cache capacities. At 64 cores, Swarm outperforms the serial versions by 43 to 117 times and the software-parallel versions by 2.7 to 18.2 times.

Swarm Resource Utilization

Swarm uses resources efficiently: it mispeculates rarely, keeps its large task and commit queues highly utilized, and adds moderate traffic and energy overheads.

Cycle breakdowns. Figure 10 shows the breakdown of core cycles for a 64-core Swarm system. Most cycles are spent executing tasks that later commit, while aborted work consumes between 1 percent (`bfs`) and 27 percent (`des`) of cycles. Finally, cores rarely stall due to full or empty queues.

Queue occupancies. Figure 11 shows the average number of task queue and commit queue entries used across the 64-core system. Both queues are often highly utilized. Commit queues can hold up to 1,024 finished tasks (64

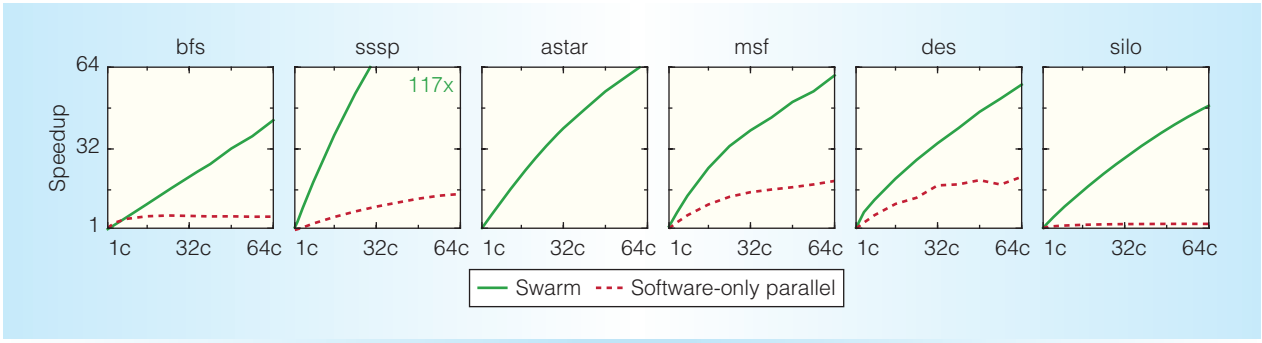

Figure 9. Speedup of Swarm and state-of-the-art software-parallel implementations from 1 to 64 cores, relative to a tuned serial implementation running on a system of the same size. At 64 cores, Swarm programs are 43 to 117 times faster than the serial versions and 2.7 to 18 times faster than software-parallel versions.

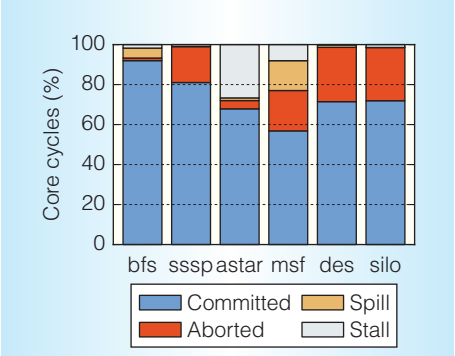

Figure 10. Breakdown of core cycles for 64-core Swarm. Each bar shows the breakdown of cycles spent running tasks that are ultimately committed, running tasks that are later aborted, spilling tasks from the hardware task queues, or stalled. Most time is spent on tasks that later commit.

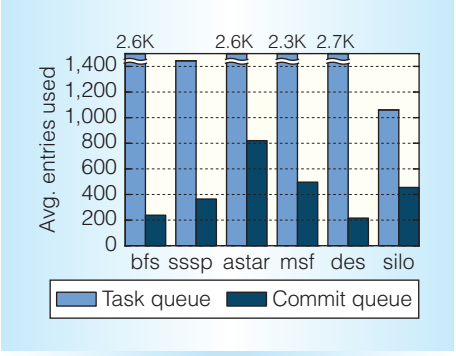

Figure 11. Average task and commit queue occupancies for 64-core Swarm. Task and commit queues are highly utilized. Swarm speculates 200 to 800 tasks ahead of the earliest active task on average, leveraging its large window of speculation.

per tile). On average, they hold from 216 in `des` to 821 in `astar`. This shows that Swarm speculates far ahead of the earliest active task, as required to extract enough parallelism. The 4,096-entry task queues are also well utilized, with average occupancies between 1,060 (`silo`) and 2,712 (`msf`) entries.

Network traffic breakdowns. Figure 12 shows the network traffic breakdown at 64 cores (16 tiles). The cumulative injection rate per tile remains well below the saturation injection rate (64 Gbytes per second). Each bar shows the contributions of memory accesses (between the L2s and L3) issued during normal execution, tasks enqueued to other tiles, traffic caused by aborts (including child abort messages and rollback memory accesses), and GVT updates sent. Task enqueues, aborts, and GVT updates increase network traffic by 15 percent on average.

Swarm's design challenges conventional wisdom in two ways. First, conventional wisdom says that order constraints limit parallelism. However, we have shown that it is possible to maintain a large speculation window efficiently, so that only true data dependences limit parallelism. Second, conventional wisdom says that speculation is wasteful, and designers should instead build nonspeculative parallel systems. However, we have shown that, for a broad class of applications, speculation unlocks abundant

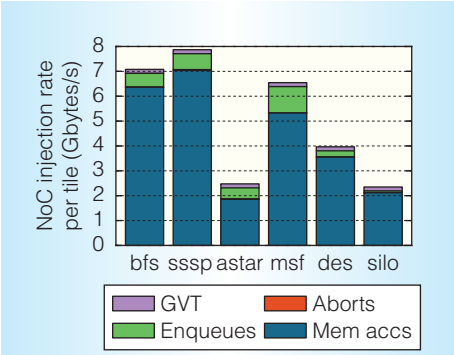

Figure 12. Breakdown of network-on-chip traffic per tile for 64-core, 16-tile Swarm. Swarm increases network traffic by a moderate amount over normal execution.

parallelism for moderate costs. Designers can trade this additional parallelism for efficiency in many ways (for example, through simpler cores or slower clocks), more than offsetting the costs of speculation. In other words, speculation can yield a net efficiency gain.

Swarm opens exciting research avenues beyond ordered irregular parallelism. For example, Swarm's techniques could be useful in making automatic or semiautomatic parallelization practical. Prior limit studies on instruction-level parallelism have shown that parallelism is plentiful in most serial applications, but exploiting it often requires extremely large speculation windows[17] (for example, more than 100,000 instructions). These speculation windows are impractical in conventional out-of-order processors. However, with 100-instruction tasks, Swarm does speculate over a window of 100,000 instructions. Can Swarm be combined with existing autoparallelization techniques to approach these large speedups? If not, can these speedups be realized with novel programming models, compiler techniques, and architectural support?

Finally, although Swarm scales well to systems with tens of cores, a key question is how much further it can scale. Although Swarm's out-of-order task execution techniques scale well, poor locality will eventually limit scaling: our current implementation spreads tasks across tiles randomly, which will cause too much data movement to be practical at higher core counts. However, we believe Swarm's techniques can help improve locality: by supporting many more tasks than cores, Swarm lets the system choose when and where to execute tasks. We could use this flexibility to map tasks across the system in a locality-aware fashion, relying on deep queues to absorb short-term load imbalance. We believe this would allow Swarm to scale much further, perhaps to multichip and multiboard implementations. We leave this and other endeavors to future work. MICRO

Acknowledgments

We thank Nathan Beckmann, Harshad Kasture, Anurag Mukkara, Li-Shiuan Peh, Po-An Tsai, Guowei Zhang, and the anonymous reviewers for their helpful feedback. M. Amber Hassaan and Donald Nguyen graciously assisted with Galois benchmarks. This work was partially supported by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grant CAREER-1452994. Mark Jeffrey was partially supported by an MIT EECS Jacobs Presidential Fellowship and an NSERC Postgraduate Scholarship.

References

1. M.A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. Unordered: A Comparison of Parallelism and Work-Efficiency in Irregular Algorithms," *Proc. 16th ACM Symp. Principles and Practice of Parallel Programming*, 2011, pp. 3–12.
2. M.C. Jeffrey et al., "A Scalable Architecture for Ordered Parallelism," *Proc. 48th Int'l Symp. Microarchitecture*, 2015, pp. 228–241.
3. K. Pingali et al., "The Tao of Parallelism in Algorithms," *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2011, pp. 12–25.
4. E.W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, 1959, pp. 269–271.
5. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, 1995, pp. 414–425.
6. L. Hammond, M. Willey, and K. Olukotun, "Data Speculation Support for a Chip Multiprocessor," *Proc. 8th Int'l Conf. Architectural*

Support for Programming Languages and Operating Systems, 1998, pp. 58–69.

7. J.G. Steffan et al., "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Ann. Int'l Symp. Computer Architecture*, 2000, pp. 1–12.
8. J. Renau et al., "Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation," *Proc. 19th Ann. Int'l Conf. Supercomputing*, 2005, pp. 179–188.
9. R. Panigrahy and S. Sharma, "Sorting and Searching Using Ternary CAMs," *IEEE Micro*, vol. 23, no. 1, 2003, pp. 44–53.
10. K.E. Moore et al., "LogTM: Log-Based Transactional Memory," *Proc. 12th Int'l Symp. High-Performance Computer Architecture*, 2006, pp. 254–265.
11. M.J. Garzarán et al., "Tradeoffs in Buffering Speculative Memory State for Thread-Level Speculation in Multiprocessors," *Proc. 9th Int'l Symp. High-Performance Computer Architecture*, 2003, pp. 191–202.
12. D. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 3, 1985, pp. 404–425.
13. A. Ferscha and S.K. Tripathi, *Parallel and Distributed Simulation of Discrete Event Systems*, tech. report, Computer Science Dept., Univ. of Maryland at College Park, 1998.
14. S. Tu et al., "Speedy Transactions in Multicore In-Memory Databases," *Proc. 24th ACM Symp. Operating Systems Principles*, 2013, pp. 18–32.
15. S. Brand and R. Bidarra, "Multi-core Scalable and Efficient Pathfinding with Parallel Ripple Search," *Computer Animation and Virtual Worlds*, vol. 23, no. 2, 2012, pp. 73–85.
16. C.E. Leiserson and T.B. Schardl, "A Work-Efficient Parallel Breadth-First Search Algorithm," *Proc. 22nd Ann. ACM Symp. Parallelism in Algorithms and Architectures*, 2010, pp. 303–314.
17. K. Ebcioğlu et al., "Optimizations and Oracle Parallelism with Dynamic Translation," *Proc. 32nd Ann. ACM/IEEE Int'l Symp. Microarchitecture*, 1999, pp. 284–295.

Mark C. Jeffrey is a PhD student in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. His research interests include parallel architectures, programming models, and speculative execution. Jeffrey received an MASc in electrical and computer engineering from the University of Toronto. He is a student member of IEEE. Contact him at mcj@csail.mit.edu.

Suvinay Subramanian is a PhD student in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. His research interests include parallel architectures and scalable multicore systems. Subramanian received an SM in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a student member of IEEE. Contact him at suvinay@csail.mit.edu.

Cong Yan is a PhD student in the Department of Computer Science and Engineering at the University of Washington. Her research interests include concurrency control protocols, program-analysis-assisted databases, and scalable multicore systems. Yan received an SM in electrical engineering and computer science from the Massachusetts Institute of Technology. She is a student member of IEEE. Contact her at congy@cs.washington.edu.

Joel Emer is a senior distinguished research scientist at Nvidia. He is also a professor of electrical engineering and computer science at the Massachusetts Institute of Technology. His research interests include spatial and parallel architectures, performance modeling, reliability analysis, and memory hierarchies. Emer received a PhD in electrical engineering from the University of Illinois. He is a Fellow of IEEE. Contact him at emer@csail.mit.edu.

Daniel Sanchez is an assistant professor in the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology. His research interests include scalable memory hierarchies, architectural support for parallelization, and architectures with quality-of-service guarantees. Sanchez received a PhD in electrical engineering from Stanford University. He is a member of IEEE. Contact him at sanchez@csail.mit.edu.