# Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism

Suvinay Subramanian* Mark C. Jeffrey* Maleen Abeydeera* Hyun Ryong Lee*
Victor A. Ying* Joel Emer*† Daniel Sanchez*
*Massachusetts Institute of Technology †NVIDIA
{suvinay,mcj,maleen,hrlee,victory,emer,sanchez}@csail.mit.edu

## ABSTRACT

Most systems that support speculative parallelization, like hardware transactional memory (HTM), do not support nested parallelism. This sacrifices substantial parallelism and precludes composing parallel algorithms. And the few HTMs that do support nested parallelism focus on parallelizing at the coarsest (shallowest) levels, incurring large overheads that squander most of their potential.

We present FRACTAL, a new execution model that supports unordered and timestamp-ordered nested parallelism. FRACTAL lets programmers seamlessly compose speculative parallel algorithms, and lets the architecture exploit parallelism at all levels. FRACTAL can parallelize a broader range of applications than prior speculative execution models. We design a FRACTAL implementation that extends the Swarm architecture and focuses on parallelizing at the finest (deepest) levels. Our approach sidesteps the issues of nested parallel HTMs and uncovers abundant fine-grain parallelism. As a result, FRACTAL outperforms prior speculative architectures by up to 88× at 256 cores.

## CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**;

## KEYWORDS

Multicore, Speculative parallelization, Nested parallelism, Fine-grain parallelism, Transactional memory, Thread-level speculation

**ACM Reference format:**
Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, Daniel Sanchez. 2017. Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080218

## 1 INTRODUCTION

Systems that support speculative parallelization, such as hardware transactional memory (HTM) or thread-level speculation (TLS), have two major benefits over non-speculative systems: they uncover abundant parallelism in many challenging applications [32, 36] and simplify parallel programming [51]. But these systems suffer from limited support for *nested speculative parallelism*, i.e., the ability to invoke a speculative parallel algorithm within another speculative parallel algorithm. This causes three problems. First, it sacrifices substantial parallelism and limits the algorithms supported by these systems. Second, it disallows composing parallel algorithms, making it hard to write modular parallel programs. Third, it biases programmers to write coarse-grain speculative tasks, which are more expensive to support in hardware.

For example, consider the problem of parallelizing a transactional database. A natural approach is to use HTM and to make each database transaction a memory transaction. Each transaction executes on a thread, and the HTM system guarantees atomicity among concurrent transactions, detecting conflicting loads and stores on the fly, and aborting transactions to avoid serializability violations.

Unfortunately, this HTM approach faces significant challenges. First, each transaction must run on a single thread, but database transactions often consist of many queries or updates that could run in parallel. The HTM approach thus sacrifices this intra-transaction, fine-grain parallelism. Second, long transactions often have large read and write sets, which make conflicts and aborts more likely. These aborts often waste many operations that were not affected by the conflict. Third, supporting large read/write sets in hardware is costly. Hardware can track small read/write sets cheaply, e.g., using private caches [32, 56] or small Bloom filters [14, 65]. But these tracking structures have limited capacity and force transactions that overflow them to serialize, even when they have no conflicts [11, 32, 65]. Beyond these problems, HTM's unordered execution semantics are insufficient for programs with ordered parallelism, where speculative tasks must appear to execute in a program-specified order [36].

The Swarm architecture [36, 37] can address some of these problems. Swarm programs consist of timestamped tasks. A task can create and enqueue child tasks with any timestamp equal to or greater than its own. Swarm guarantees that tasks appear to execute in timestamp order. To scale, Swarm executes tasks speculatively and out of order. Swarm's microarchitecture focuses on supporting tasks as small as a few tens of instructions efficiently, including hardware support for speculative scheduling and a large speculation window.

By exposing timestamps to programs, Swarm can parallelize more algorithms than prior ordered speculation techniques, like TLS; Swarm also supports unordered, HTM-style execution. As a result, Swarm often uncovers abundant fine-grain parallelism. But Swarm's software-visible timestamps can only convey very limited forms of nested parallelism, and they cause two key issues in this regard (Sec. 2). Timestamps make nested algorithms *hard to compose*, as algorithms at different nesting levels must agree on a common meaning for the timestamp. Timestamps also *over-serialize* nested algorithms, as they impose more order constraints than needed.

For instance, in the example above, Swarm can be used to break each database transaction into many small, ordered tasks. This exploits intra-transaction parallelism, and, at 256 cores, it is 21× faster than running operations within each transaction serially (Sec. 2.2).

However, to maintain atomicity among database transactions, the programmer must needlessly order database transactions and must carefully assign timestamps to tasks within each transaction.

These problems are far from specific to database transactions. In general, large programs have speculative parallelism at multiple levels and often intermix ordered and unordered algorithms. Speculative architectures should support composition of ordered and unordered algorithms to convey all this nested parallelism without undue serialization.

We present two main contributions that achieve these goals. Our first contribution is **FRACTAL**, a new execution model for nested speculative parallelism. FRACTAL programs consist of tasks located in a hierarchy of nested *domains*. Within each domain, tasks can be ordered or unordered. Any task can create a new subdomain and enqueue new tasks in that subdomain. All tasks in a domain appear to execute atomically with respect to tasks outside the domain.

FRACTAL allows seamless composition of ordered and unordered nested parallelism. In the above example, each database transaction starts as a single task that runs in an unordered, root domain. Each of these unordered tasks creates an ordered subdomain in which it enqueues tasks for the different operations within the transaction. In the event of a conflict between tasks in two different transactions, FRACTAL selectively aborts conflicting tasks, rather than aborting all tasks in any one transaction. In fact, other tasks from the two transactions may continue to execute in parallel.

Our second contribution is a simple implementation of FRACTAL that builds on Swarm and supports arbitrary nesting levels cheaply (Sec. 4). Our implementation focuses on extracting parallelism at the finest (deepest) levels first. This is in stark contrast with current HTMs. Most HTMs only support serial execution of nested transactions, forgoing intra-transaction parallelism. A few HTMs support parallel nested transactions [6, 62], but they parallelize at the coarsest levels, suffer from subtle deadlock and livelock conditions, and impose large overheads because they merge the speculative state of nested transactions [5, 6]. The FRACTAL execution model lets our implementation avoid these problems. Beyond exploiting more parallelism, focusing on fine-grain tasks reduces the hardware costs of speculative execution.

We demonstrate FRACTAL's performance and programmability benefits through several case studies (Sec. 2) and a broad evaluation (Sec. 6). FRACTAL uncovers abundant fine-grain parallelism on large programs. For example, ports of the STAMP benchmark suite to FRACTAL outperform baseline HTM implementations by up to 88× at 256 cores. As a result, while several of the original STAMP benchmarks cannot reach even 10× scaling, FRACTAL makes all STAMP benchmarks scale well to 256 cores.

## 2 MOTIVATION

We motivate FRACTAL through three case studies that highlight its key benefits: uncovering abundant parallelism, improving programmability, and avoiding over-serialization. Since FRACTAL subsumes prior speculative execution models (HTM, TLS, and Swarm), all case studies use the FRACTAL architecture (Sec. 4), and we compare applications written in FRACTAL vs. other execution models. This approach lets us focus on the effect of different FRACTAL features. Our implementation does not add overheads to programs that do not use FRACTAL's features.

### 2.1 Fractal uncovers abundant parallelism

Consider the maxflow problem, which finds the maximum amount of flow that can be pushed from a source to a sink node in a network (a graph with directed edges labeled with capacities). Push-relabel is a fast and widely used maxflow algorithm [18], but it is hard to parallelize [8, 48]. Push-relabel tags each node with a height. It initially gives heights of 0 to the sink, $N$ (the number of nodes) to the source, and 1 to every other node. Nodes are temporarily allowed to have excess flow, i.e., have more incoming flow than outgoing flow. Nodes with excess flow are considered *active* and can push this flow to lower-height nodes. The algorithm processes one active node at a time, attempting to push flow to neighbor nodes and potentially making them active. When an active node cannot push its excess flow, it increases its height to the minimum value that allows pushing flow to a neighbor (this is called a relabel). The algorithm processes active nodes in arbitrary order until no active nodes are left.

To be efficient, push-relabel must use a heuristic that periodically recomputes node heights. Global relabeling [18] is a commonly used heuristic that updates many node heights by performing a breadth-first search on a subset of the graph. Global relabeling takes a significant fraction of the total work, typically 10–40% of instructions [3].

Since push-relabel can process active nodes in an arbitrary order, it can be parallelized using transactional tasks of two types [48, 49]. An **active-node** task processes a single node, and may enqueue other tasks to process newly-activated nodes. A **global-relabel** task performs a global relabel operation. Each task must run atomically, since tasks access data from multiple neighbors and must observe a consistent state. We call this implementation `maxflow-flat`.

We simulate `maxflow-flat` on systems of up to 256 cores. (See Sec. 5 for methodology details.) At 256 cores, `maxflow-flat` scales to 4.9× only. Fig. 1a illustrates the reason for this limited speedup: while **active-node** tasks are short, each **global-relabel** task is long, and queries and updates many nodes. When a **global-relabel** task runs, it conflicts with and serializes many **active-node** tasks.

Fortunately, each **global-relabel** task performs a breadth-first search, which has plentiful *ordered* speculative parallelism. FRACTAL lets us exploit this nested parallelism, running the breadth-first

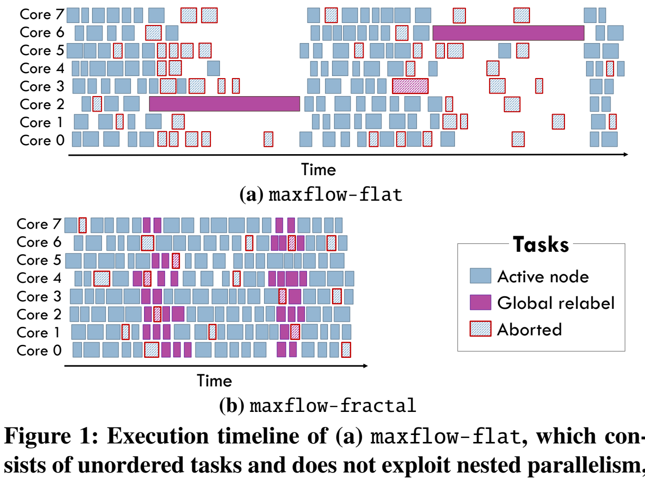


**Figure 1: Execution timeline of (a) `maxflow-flat`, which consists of unordered tasks and does not exploit nested parallelism, and (b) `maxflow-fractal`, which exploits the nested ordered parallelism within global relabel.**

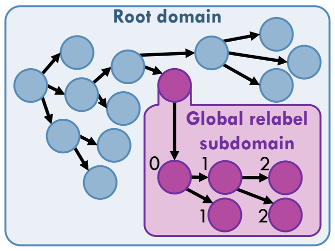


**Figure 2: In `maxflow-fractal`, each global-relabel task creates an ordered subdomain.**

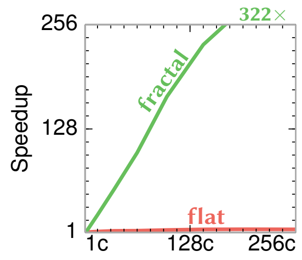


**Figure 3: Speedup of different `maxflow` versions on 1–256 cores.**

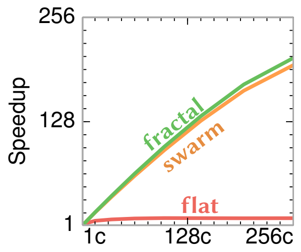


**Figure 4: Speedup of `silo` versions on 1–256 cores.**

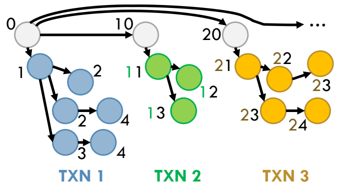


**Figure 5: `silo-swarm` uses disjoint timestamp ranges for different database transactions, sacrificing composability.**

search in parallel while maintaining its atomicity with respect to other active-node tasks. To achieve this, we develop a `maxflow-fractal` implementation where each global-relabel task creates an ordered subdomain, in which it executes a parallel breadth-first search using fine-grain ordered tasks, as shown in Fig. 2. A global-relabel task and its subdomain appear as a single atomic unit with respect to other tasks in the (unordered) root domain. Fig. 1b illustrates how this improves parallelism and efficiency. As a result, Fig. 3 shows that `maxflow-fractal` achieves a speedup of 322× at 256 cores (over `maxflow-flat` on one core).

FRACTAL is the first architecture that effectively exploits maxflow's fine-grain nested parallelism: neither HTM, nor TLS, nor Swarm can support the combination of unordered and ordered parallelism maxflow has. Prior software-parallel push-relabel algorithms attempted to exploit this fine-grain parallelism [3, 48, 49], but the overheads of software speculation and scheduling negated the benefits of additional parallelism (in `maxflow-fractal`, each task is 373 cycles on average). We also evaluated two state-of-the-art software implementations: `prsn` [8] and Galois [48]. On 1–256 cores, they achieve maximum speedups of only 4.9× and 8.3× over `maxflow-flat` at one core, respectively.

## 2.2 Fractal eases parallel programming

Beyond improving performance, FRACTAL's support for nested parallelism eases parallel programming because it enables *parallel composition*. Programmers can write multiple self-contained, modular parallel algorithms and compose them without sacrificing performance: when a parallel algorithm invokes another parallel algorithm, FRACTAL can exploit parallelism at both caller and callee.

In the previous case study, only FRACTAL was able to uncover nested parallelism. In some applications, prior architectures can also exploit the nested parallelism that FRACTAL uncovers, but they do so at the expense of composability.

Consider the transactional database example from Sec. 1. Conventional HTMs run each database transaction in a single thread, and exploit coarse-grain inter-transaction parallelism only. But each database transaction has plentiful ordered parallelism. FRACTAL can exploit both inter- and intra-transaction parallelism by running each transaction in its own ordered subdomain, just as each global relabel runs in its own ordered subdomain in Fig. 2. We apply both approaches to the `silo` in-memory database [61]. Fig. 4 shows that, at 256 cores, `silo-fractal` scales to 206×, while `silo-flat` scales to 9.7× only, 21× slower than `silo-fractal`.

Fig. 4 also shows that `silo-swarm`, the Swarm version of `silo`, achieves similar performance to `silo-fractal` (`silo-swarm` is 4.5% slower). Fig. 5 illustrates `silo-swarm`'s implementation: the transaction-launching code assigns disjoint timestamp ranges to transactions (10 contiguous timestamps per transaction in Fig. 5), and each transaction enqueues tasks only within this range (e.g., 10–19 for TXN 2 in Fig. 5). `silo-swarm` uses the same fine-grain tasks as `silo-fractal`, exposing plentiful parallelism and reducing the penalty of conflicts [36]. For example, in Fig. 5, if the tasks at timestamps 13 and 24 conflict, only one task must abort, rather than any whole transaction.

Since Swarm does not provide architectural support for nested parallelism, approaching FRACTAL's performance comes at the expense of composability. `silo-swarm` *couples* the transaction-launching code and the code within each transaction: both modules must know the number of tasks per transaction, so that they can agree on the semantics of each timestamp. Moreover, a fixed-size timestamp makes it hard to allocate sufficient timestamp ranges in complex applications with many nesting levels or where the number of tasks in each level is dynamically determined. FRACTAL avoids these issues by providing direct support for nested parallelism.

Prior HTMs have supported composable nested parallel transactions, but they suffer from deadlock and livelock conditions, impose large overheads, and sacrifice most of the benefits of fine-grain parallelism because each nested transaction merges its speculative state with its parent's [5, 6]. We compare FRACTAL and parallel nesting HTMs in detail in Sec. 7, after discussing FRACTAL's implementation. Beyond these issues, parallel nesting HTMs do not support ordered parallelism, so they would not help `maxflow` or `silo`.

## 2.3 Fractal avoids over-serialization

Beyond forgoing composability, supporting fine-grain parallelism through manually-specified ordering can cause over-serialization.

Consider the maximal independent set algorithm (`mis`), which, given a graph, finds a set of nodes $S$ such that no two nodes in $S$ are adjacent, and each node not in $S$ is adjacent to some node in $S$.

`mis` can be easily parallelized with unordered, atomic tasks [54]. We call this implementation `mis-flat`. Each task operates on a node and its neighbors. If the node has not yet been visited, the task visits both the node and its neighbors, adding the node to the set and marking its neighbors as excluded from the set. `mis-flat` creates one task for each node in the graph, and finishes when all these tasks have executed. Fig. 6 shows that, on an R-MAT graph with 8 million nodes and 168 million edges, `mis-flat` scales to 98× at 256 cores.

mis-flat misses a source of nested parallelism: when a node is added to the set, its neighbors may be visited and excluded in parallel. This yields great benefits when nodes have many neighbors. mis-fractal defines two task types: include and exclude. An include task checks whether a node has already been visited. If it has not, it adds the node to the set and creates an unordered subdomain to run exclude tasks for the node's neighbors. An exclude task permanently excludes a node from the set. Domains guarantee a node and its neighbors are visited atomically while allowing many tasks of both types to run in parallel. Fig. 6 shows that mis-fractal scales to 145× at 256 cores, 48% faster than mis-flat.

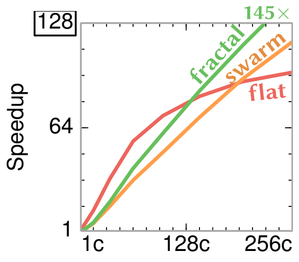


**Figure 6: Speedup of different mis versions on 1–256 cores.**

Swarm cannot exploit this parallelism as effectively. Swarm can only guarantee atomicity for groups of tasks if the program specifies a total order among groups (as in silo). We follow this approach to implement mis-swarm: every include task is assigned a unique timestamp, and it shares its timestamp with any exclude tasks it enqueues. This imposes more order constraints than mis-fractal, where there is no order among tasks in the root domain. Fig. 6 shows that mis-swarm scales to 117×, 24% slower than mis-fractal, as unnecessary order constraints cause more aborted work.[1]

In summary, conveying the atomicity needs of nested parallelism through a fixed order limits parallel execution. FRACTAL allows programs to convey nested parallelism without undue order constraints.

## 3 FRACTAL EXECUTION MODEL

Fig. 7 illustrates the key elements of the FRACTAL execution model. FRACTAL programs consist of *tasks* in a hierarchy of nested *domains*. Each task may access arbitrary data, and may create child tasks as it finds new work to do. For example, in Fig. 7 task C creates children D and E. When each task is created, it is enqueued to a specific domain.

**Semantics within a domain:** Each domain provides either unordered or timestamp-ordered execution semantics. In an unordered domain, FRACTAL chooses an arbitrary order among tasks that respects parent-child dependences, i.e., children are ordered after their parents. For example, in Fig. 7, task C's children D and E must appear to run after C, but task D can appear to run either before or after task E. These semantics are similar to TM's: all tasks execute atomically and in isolation.

In an ordered domain, each task has a program-specified timestamp. A task can enqueue child tasks to the same domain with any timestamp equal to or greater than its own. FRACTAL guarantees that tasks appear to run in increasing timestamp order. If multiple tasks have the same timestamp, FRACTAL arbitrarily chooses an order among them. This order always respects parent-child dependences. Timestamps let programs convey their specific order requirements, e.g., the order that events need to run in a simulator. For example,

[1]mis-swarm's order constraints make it deterministic, which some users may find desirable [9, 23].

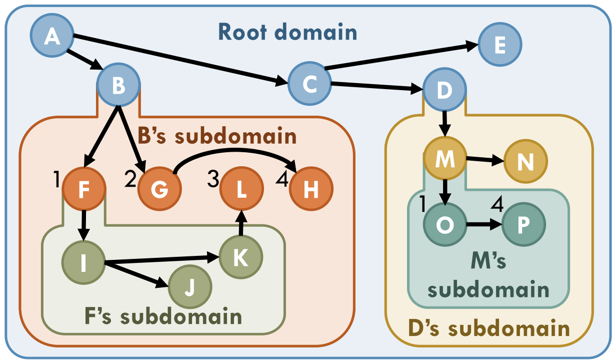


**Figure 7: Elements of the FRACTAL execution model. Arrows point from parent to child tasks. Parents enqueue their children into ordered domains where tasks have timestamps, such as A's and M's subdomains, or unordered domains, such as the other three domains.**

in Fig. 7, the timestamps of tasks F, G, L, and H ensure they appear to run in that fixed order. These semantics are the same as Swarm's [36].

**Semantics across domains:** Each task can create a single subdomain and enqueue tasks into it. For example, in Fig. 7, task B creates a new subdomain and enqueues F and G into it. These tasks may themselves create their own subdomains. For example, F creates a subdomain and enqueues I into it.

FRACTAL provides strong atomicity guarantees across domains to allow parallel composition of speculative algorithms. All tasks in a domain appear to execute after the task that creates the domain and are not interleaved with tasks outside their domain. In other words, any non-root domain together with its creator appears to execute as a *single atomic unit* in isolation. For example, since F is ordered before G in B's subdomain, all tasks in F's subdomain (I, J, and K) must appear to execute immediately after F and before G. Furthermore, although no task in B's subdomain is ordered with respect to any task in D's subdomain, tasks in B's and D's subdomains are guaranteed not to be interleaved.

Tasks may also enqueue child tasks to their immediate enclosing domain, or *superdomain*. For example, in Fig. 7, K in F's subdomain enqueues L to B's subdomain. This lets a task delegate enqueuing future work to descendants within the subdomain it creates. A task cannot enqueue children to any domain beyond the domain it belongs to, its superdomain, and the single subdomain it may create.

### 3.1 Programming interface

We first expose FRACTAL's features through a simple low-level C++ interface, then complement it with a high-level, OpenMP-style interface that makes it easier to write FRACTAL applications.

**Low-level interface:** Listing 1 illustrates the key features of the low-level FRACTAL interface by showing the implementation of the mis-fractal tasks described in Sec. 2.3. A task is described by its function, arguments, and ordering properties. Task functions can take arbitrary arguments but do not return values. Tasks create children by calling one of three enqueue functions with the appropriate task function and arguments: `fractal::enqueue` places the child task in the same domain as the caller, `fractal::enqueue_sub` places the

```
void exclude(Node& n) {
  n.state = EXCLUDED;
}

void include(Node& n) {
  if (n.state == UNVISITED) {
    n.state = INCLUDED;
    fractal::create_subdomain(UNORDERED);
    for (Node& ngh: n.neighbors)
      fractal::enqueue_sub(exclude, ngh);
  }
}
```

**Listing 1: FRACTAL implementation of `mis` tasks.**

```
void include(Node& n) {
  if (n.state == UNVISITED) {
    n.state = INCLUDED;
    forall (Node& ngh: n.neighbors)
      ngh.state = EXCLUDED;
  }
}
```

**Listing 2: Pseudocode for FRACTAL implementation of `mis`'s `include` using the high-level interface.**

child in the caller's subdomain, and `fractal::enqueue_super` places the child in the caller's superdomain. If the destination domain is ordered, the enqueuing function also takes the child task's timestamp. This isn't the case in Listing 1, as `mis` is unordered.

Before calling `fractal::enqueue_sub` to place tasks in a subdomain, a task must call `fractal::create_subdomain` exactly once to specify the subdomain's ordering semantics: unordered, or ordered with 32- or 64-bit timestamps. In Listing 1, each `include` task may create an unordered subdomain to atomically run `exclude` tasks for all its neighbors. The initialization code (not shown) creates an `include` task for every node in an unordered root domain.

Task enqueue functions also take one optional argument, a spatial hint [35], which is an integer that abstractly indicates what data the task is likely to access. Hints aid the system in performing locality-aware task mapping and load balancing. Hints are orthogonal to FRACTAL. We adopt them because we study systems of up to 256 cores, and several of our benchmarks suffer from poor locality without hints, which limits their scalability beyond tens of cores.

**High-level interface:** Although our low-level interface is simple, breaking straight-line code into many task functions can be tedious. To ease this burden, we implement a high-level interface in the style of OpenMP and OpenTM [7]. Table 1 details its main constructs, and Listing 2 shows it in action with *pseudocode* for `include`. Nested parallelism is expressed using `forall`, which automatically creates an unordered subdomain and enqueues each loop iteration as a separate task. This avoids breaking code into small functions like `exclude`. These constructs can be arbitrarily nested. Our actual syntax is slightly more complicated because we do not modify the compiler, and we implement these constructs using macros.[2]

## 4 FRACTAL IMPLEMENTATION

Our FRACTAL implementation seeks three desirable properties. First, the architecture should perform *fine-grain speculation*, carrying out conflict resolution and commits at the level of individual tasks, not complete domains. This avoids the granularity issues of nested parallel HTMs (Sec. 7). Second, *creating a domain should be cheap*, as domains with few tasks are common (e.g., `mis` in Sec. 2.3). Third, while the architecture should support unbounded nesting depth to enable software composition, parallelism compounds quickly with depth, so *hardware only needs to support a few concurrent depths*.

To meet these objectives, our FRACTAL implementation builds on Swarm, and dynamically chooses a task commit order that satisfies FRACTAL's semantics. We first describe the Swarm microarchitecture, then introduce the modifications needed to support FRACTAL.

| Function | Description |
|---|---|
| `forall` | Atomic unordred loop. Enqueues each iteration as as a task in a new unordered subdomain. |
| `forall_ordered` | Atomic ordered loop. Enqueues tasks to a new ordered subdomain, using the iteration index as a timestamp. |
| `forall_reduce` | Atomic unordered loop with a reduction variable. |
| `forall_reduce_ordered` | Atomic ordered loop with a reduction variable. |
| `parallel` | Execute multiple code blocks as parallel tasks. |
| `parallel_reduce` | Execute multiple code blocks as parallel tasks, followed by a reduction. |
| `enqueue_all` | Enqueues a sequence of tasks with the same (or no) timestamp. |
| `enqueue_all_ordered` | Enqueues a sequence of tasks with a range of timestamps. |
| `task` | Starts a new task in the middle of a function. Implicitly encapsulates the rest of the function into a lambda, then enqueues it. Useful to break long functions into smaller tasks. |
| `callcc` | Call with current continuation [59]. Allows calling a function that might enqueue tasks, returning control to the caller by invoking its continuation. The continuation runs as a separate task. |

**Table 1: High-level interface functions.**

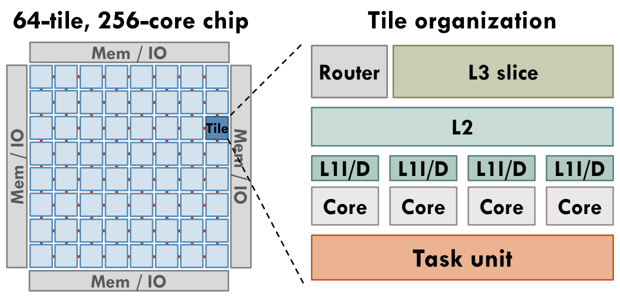


**Figure 8: 256-core Swarm chip and tile configuration.**

### 4.1 Baseline Swarm microarchitecture

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 unfinished task. Swarm introduces modest changes to a tiled, cache-coherent multicore, shown in Fig. 8. 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 hardware efficiently supports fine-grain tasks and a large speculation window through four main mechanisms: low-overhead hardware task management, large task queues, scalable data-dependence speculation techniques, and high-throughput ordered commits.

[2] The difference between the pseudocode in Listing 2 and our actual code is that we have to tag the end of control blocks, i.e., using `forall_begin(...) {...} forall_end();`. This could be avoided with compiler support, as in OpenMP.

**Hardware task management:** Tasks create child tasks and enqueue them to a tile using an `enqueue` instruction with arguments stored in registers. Each tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Each task is represented by a task descriptor that contains its function pointer, arguments, timestamp, and spatial hint [35].

Cores dequeue tasks for execution from the local task unit. Task units can dispatch any available task to cores, however distant in program order. Dequeuing initiates speculative execution at the task's function pointer and makes the task's timestamp and arguments available in registers. A core stalls if there is no task available.

**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., 16384 tasks in the 256-core chip in Fig. 8). Nevertheless, because programs can enqueue tasks with arbitrary timestamps, task and commit queues can fill up. Tasks that have not been dequeued and whose parent has committed can be *spilled* to memory to free task queue entries. Queue resource exhaustion can also be handled by either stalling the enqueuer or aborting higher-timestamp tasks to free space [36].

**Scalable data-dependence speculation:** Swarm uses eager (undo-log-based) version management and eager conflict detection using Bloom filters, similar to LogTM-SE [65]. Swarm always forwards still-speculative data read by a later task. On a conflict, Swarm aborts only descendants and data-dependent tasks.

**High-throughput ordered commits:** Finally, Swarm adapts the virtual time algorithm [34] to achieve high-throughput ordered commits. Tiles periodically communicate with a central arbiter (e.g., every 200 cycles) to discover the earliest unfinished task in the system. All tasks that precede this earliest unfinished task can safely commit. This scheme can sustain multiple task commits per cycle on average, efficiently supporting ordered tasks as short as a few tens of cycles.

Swarm maintains a consistent order among tasks by giving a unique *virtual time* (VT) to each task when it is dispatched. Swarm VTs are 128-bit integers that extend the 64-bit program-assigned timestamp with a 64-bit *tiebreaker*. This tiebreaker is the concatenation of the *dispatch cycle* and *tile id*, as shown in Fig. 9. Thus, Swarm VTs break ties among same-timestamp tasks sensibly (prioritizing older tasks), and they satisfy Swarm's semantics (they order a child task after its parent, since the child is always dispatched at a later cycle). However, Fractal needs a different schema to match its semantics.

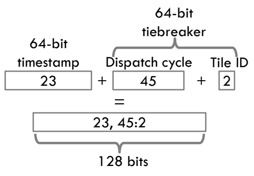


**Figure 9: Swarm VT construction.**

## 4.2 Fractal virtual time

Fractal assigns a *fractal virtual time* (fractal VT) to each task. This fractal VT is the concatenation of one or more *domain virtual times* (domain VTs).

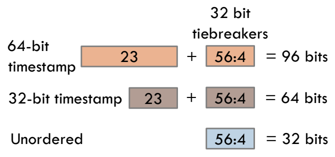


**Figure 10: Domain VT formats.**

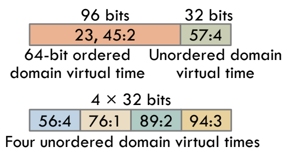

96 bits
32 bits
23, 45:2
57:4
64-bit ordered domain virtual time
Unordered domain virtual time
4 × 32 bits
56:4
76:1
89:2
94:3
Four unordered domain virtual times

**Figure 11: Example 128-bit fractal VTs.**

**Domain VTs** order all tasks in a domain and are constructed similarly to Swarm VTs. In an ordered domain, each task's domain VT is the concatenation of its 32- or 64-bit timestamp and a tiebreaker. In an unordered domain, tasks do not have timestamps, so each task's domain VT is just a tiebreaker, assigned at dispatch time.

Fractal uses 32-bit rather than 64-bit tiebreakers for efficiency. As in Swarm, each tiebreaker is the concatenation of *dispatch cycle* and *tile id*, which orders parents before children. While 32-bit tiebreakers are efficient, they can wrap around. Sec. 4.4 discusses how Fractal handles wrap-arounds. Fig. 10 illustrates the possible formats of a domain VT, which can take 32, 64, or 96 bits.

**Fractal VTs** enforce a total order among tasks in the system. This order satisfies Fractal's semantics across domains: all tasks within each domain are ordered immediately after the domain's creator and before any other tasks outside the domain. These semantics can be implemented with two simple rules. First, the fractal VT of a task in the root domain is just its root domain VT. Second, the fractal VT of any other task is equal to its domain VT appended to the fractal VT of the task that created its domain. Fig. 11 shows some example fractal VT formats. A task's fractal VT is thus made up of one domain VT for each enclosing domain. Two fractal VTs can be compared with a natural lexicographic comparison.

Fractal VTs are easy to support in hardware. We use a fixed-width field in the task descriptor to store each fractal VT, 128 bits in our implementation. Fractal VTs smaller than 128 bits are right-padded with zeros. This fixed-width format makes comparing fractal VTs easy, requiring conventional 128-bit comparators. With a 128-bit budget, Fractal hardware can support up to four levels of nesting, depending on the sizes of domain VTs. Sec. 4.3 describes how to support levels beyond those that can be represented in 128 bits.

Fig. 12 shows fractal VTs in a system with three domains: an unordered root domain, B's subdomain (ordered with 64-bit timestamps), and D's subdomain (unordered). Idle tasks do not have tiebreakers, which are assigned on dispatch. Any two dispatched tasks can be ordered by comparing their fractal VTs. For example, F (in B's subdomain) is ordered after B, but before M (in D's subdomain). Fractal performs fine-grain speculation by using the

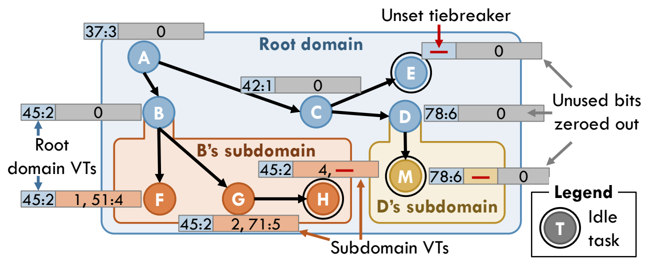
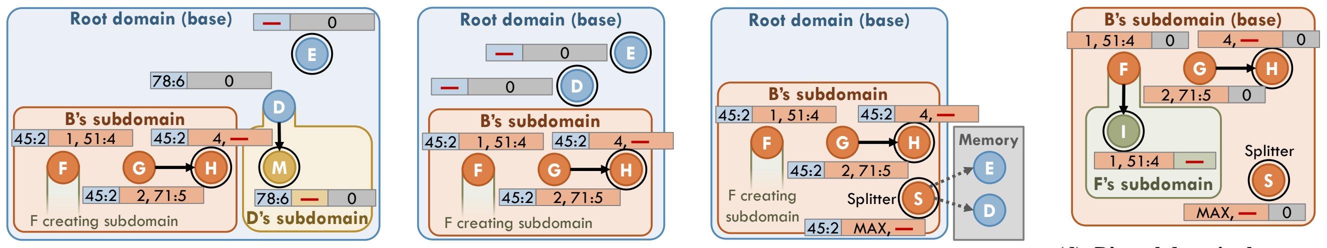


**Figure 12: Fractal VTs in action.**



(a) After B commits.

(b) Base-domain tasks abort.

(c) Base-domain tasks are spilled.

(d) B's subdomain becomes the base domain.

**Figure 13: Starting from Fig. 12, zooming in allows F to create and enqueue to a subdomain by shifting fractal VTs.**

total order among running tasks to commit and resolve conflicts at the level of individual tasks. For example, although all tasks in B's subdomain must stay atomic with respect to tasks in any other domain, FRACTAL can commit tasks B and F individually, without waiting for G and H to finish. FRACTAL guarantees that B's subdomain executes atomically because G and H are ordered before any of the remaining uncommitted tasks.

Fractal VTs also make it trivial to create a new domain. In hardware, enqueuing to a subdomain simply requires including the parent's full fractal VT in the child's task descriptor. For instance, when B enqueues F in Fig. 12, it tags F with (45:2; 1)—B's fractal VT (45:2) followed by F's timestamp (1). Similarly, enqueues to the same domain use the enqueuer's fractal VT without its final domain VT (e.g., when A enqueues C, C's fractal VT uses no more bits than A's), and enqueues to the superdomain use the enqueuer's fractal VT without its final two domain VTs.

In summary, fractal VTs capture all the information needed for ordering and task enqueues, so these operations do not rely on centralized structures. Moreover, the rules of fractal VT construction automatically enforce FRACTAL's semantics across domains while performing speculation only at the level of fine-grain tasks—no tracking is done at the level of whole domains.

## 4.3 Supporting unbounded nesting

Large applications may consist of parallel algorithms nested with arbitrary depth. FRACTAL supports this unbounded nesting depth by spilling tasks from shallower domains to memory. These spilled tasks are filled back into the system after deeper domains finish. This process, which we call *zooming*, is conceptually similar to the stack spill-fill mechanism in architectures with register windows [30]. Zooming in allows FRACTAL to continue fine-grain speculation among tasks in deeper domains, without requiring additional structures to track speculative state. Note that, although zooming is involved, it imposes negligible overheads: zooming is not needed in our full applications (which use two nesting levels), and it happens infrequently in microbenchmarks (Sec. 6.3).

**Zooming in** spills tasks from the shallowest active domain, which we call the *base domain*, to make space for deeper domains. Suppose that, in Fig. 12, F in B's subdomain wants to create an unordered subdomain and enqueue a child into it. The child's fractal VT must include a new subdomain VT, but no bits are available to the right of F's fractal VT. To solve this, F issues a *zoom-in request* to the central arbiter with its fractal VT.

Fig. 13 illustrates the actions taken during a zoom-in. To avoid priority inversion, the task that requests the zoom-in waits until the base domain task that shares its base domain VT commits. This guarantees that no active base domain tasks precede the requesting task. In our example, F waits until B commits. Fig. 13a shows the state of the system at this point—note that F and all other tasks in B's subdomain precede the remaining base-domain tasks. The arbiter broadcasts the zoom-in request and saves any timestamp component of the base domain VT to an in-memory stack. In Fig. 13a, the base domain is unordered so there is no timestamp for the arbiter to save.

Each zoom-in proceeds in two steps. First, all tasks in the base domain are spilled to memory. For simplicity, speculative state is never spilled. Instead, any base-domain tasks that are running or have finished are aborted first, which recursively aborts and eliminates their descendants. Fig. 13b shows the state of the system after these aborts. Note how D's abort eliminates M and D's entire subdomain. Although spilling tasks to memory is complex, it reuses the spilling mechanism already present in Swarm [36]: task units dispatch *coalescer* tasks that remove base-domain tasks from task queues, store them in memory, and enqueue a *splitter* task that will later re-enqueue the spilled tasks. The splitter task is deprioritized relative to all regular tasks. Fig. 13c shows the state of the system once all base-domain tasks have been spilled. A new splitter task, S, will re-enqueue D and E to the root domain when it runs.

In the second step of zooming in, the system turns the outermost subdomain into the base domain. At this point, all tasks belong to one subdomain (B's subdomain in our example), so their fractal VTs all begin with the same base domain VT. This common prefix may be eliminated while preserving order relations. Each tile walks its task queues and modifies the fractal VTs of all tasks by shifting out the common base domain VT. Each tile also modifies its canary VTs, which enable the L2 to filter conflict checks [36]. Overall, this requires modifying a few tens to hundreds of fractal VTs per tile (in our implementation, up to 256 in the task queue and up to 128 canaries). Fig. 13d shows the state of the system after zooming in. B's subdomain has become the base domain. This process has freed 32 bits of fractal VT, so F can enqueue I into its new subdomain.

**Zooming out** reverses the effects of zooming in. It is triggered when a task in the base domain wants to enqueue to its superdomain. The enqueuing task first waits until all tasks preceding it have committed. Then, it sends a *zoom-out request* to the central arbiter with its fractal VT. If the previous base domain was ordered, the central arbiter pops a timestamp from its stack to broadcast with the zoom-out request.

Zooming out restores the previous base domain: Each tile walks its task queues, right-shifting each fractal VT and adding back the base domain timestamp, if any. The restored base domain VT has its tiebreaker set to zero, but this does not change any order relations because the domain from which we are zooming out contains all the earliest active tasks.

**Avoiding quiescence:** As explained so far, the system would have to be completely quiesced while fractal VTs are being shifted. This overhead is small—a few hundred cycles—but introducing mechanisms to quiesce the whole system would add complexity. Instead, we use an alternating-bit protocol [60] to let tasks continue running while fractal VTs are modified. Each fractal VT entry in the system has an extra bit that is flipped on each zoom in/out operation. When the bits of two fractal VTs being compared differ, one of them is shifted appropriately to perform the comparison.

## 4.4 Handling tiebreaker wrap-arounds

Using 32-bit tiebreakers makes fractal VTs compact, but causes tiebreakers to wrap around every few tens of milliseconds. Since domains can exist for long periods of time, the range of existing tiebreakers must be compacted to make room for new ones. When tiebreakers are about to wrap around, the system walks every fractal VT and performs the following actions:

(1) Subtract $2^{31}$ (half the range) with saturate-to-0 from each tiebreaker in the fractal VT (i.e., flip the MSB from 1 to 0, or zero all the bits if the MSB was 0).
(2) If a task's *final* tiebreaker is 0 after subtraction *and* the task is not the earliest unfinished task, abort it.

When this process finishes, all tiebreakers are $< 2^{31}$, so the system continues assigning tiebreakers from $2^{31}$.

This exploits the property that, if the task that created a domain precedes all other active tasks, its tiebreaker can be set to zero without affecting order relations. If the task is aborted because its tiebreaker is set to 0, any subdomain it created will be squashed. In practice, we find this has no effect on performance, because, to be aborted, a task would have to remain speculative for far longer than we observe in any benchmark.

## 4.5 Putting it all together

Our FRACTAL implementation adds small hardware overheads over Swarm. (Swarm itself imposes modest overheads to implement speculative execution [36].) Each fractal VT consumes five additional bits beyond Swarm's 128: four to encode its format (14 possibilities), and one for the alternating-bit protocol. This adds storage overheads of 240 bytes per 4-core tile. FRACTAL also adds simple logic to each tile to walk and modify fractal VTs—for zooming and tiebreaker wrap-arounds—and adds a shifter to fractal VT comparators to handle the alternating-bit protocol.

FRACTAL makes small changes to the ISA: it modifies the `enqueue` instruction and adds a `create_subdomain` instruction. Task enqueue messages carry a fractal VT without the final tiebreaker (up to 96+5 bits) compared to the 64-bit timestamp in Swarm.

Finally, in our implementation, zoom-in/out requests and tiebreaker wrap-arounds are handled by the global virtual time arbiter (the unit that runs the ordered-commit protocol). This adds a few message types between this arbiter and the tiles to carry out the steps in each of these operations. The arbiter must manage a simple in-memory stack to save and restore base domain timestamps.

| | |
|---|---|
| **Cores** | 256 cores in 64 tiles (4 cores/tile), 2 GHz, x86-64 ISA; 8B-wide ifetch, 2-level bpred with 256×9-bit BHSRs + 512×2-bit PHT, single-issue in-order scoreboarded (stall-on-use), functional unit latencies as in Nehalem [52], 4-entry load and store buffers |
| **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<br>32 lines per fractal VT canary |
| **L3 cache** | 64 MB, shared, static NUCA [38] (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 [64]) |
| **Main mem** | 4 controllers at chip edges, 120-cycle latency |
| **Queues** | 64 task queue entries/core (16384 total),<br>16 commit queue entries/core (4096 total),<br>128-bit fractal VTs |
| **FRACTAL instructions** | 5 cycles per `enqueue/dequeue/finish_task`<br>2 cycles per `create_subdomain` instruction |
| **Scheduler** | Spatial hints with load balancing [35] |
| **Conflicts** | 2 Kbit 8-way Bloom filters, $H_3$ hash functions [13]<br>Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue |
| **Commits** | Tiles send updates to GVT arbiter every 200 cycles |
| **Spills** | Coalescers fire when a task queue is 85% full<br>Coalescers spill up to 15 tasks each |

**Table 2: Configuration of the 256-core system.**

| | Application | Input | 1-core runtime (B cycles) |
|---|---|---|---|
| Swarm | **color** [33] | com-youtube [39] | 0.968 |
| | **msf** [54] | kron_g500-logn16 [4, 21] | 0.717 |
| | **silo** [61] | TPC-C, 4 whs, 32 Ktxns | 2.98 |
| STAMP [42] | **ssca2** | -s15 -i1.0 -u1.0 -l6 -p6 | 10.6 |
| | **vacation** | -n4 -q60 -u90 -r1048576 -t262144 | 4.31 |
| | **genome** | -g4096 -s48 -n1048576 | 2.26 |
| | **kmeans** | -m40 -n40 -i rand-n16384-d24-c16 | 8.75 |
| | **intruder** | -a10 -l64 -s32768 | 2.12 |
| | **yada** | -a15 -i ttimeu100000.2 | 3.41 |
| | **labyrinth** | random-x128-y128-z5-n128 | 4.41 |
| | **bayes** | -v32 -r4096 -n10 -p40 -i2 -e8 -s1 | 8.81 |
| | **maxflow** [8] | rmf-wide [18, 29], 65 K nodes, 314 K edges | 16.7 |
| | **mis** [54] | R-MAT [15], 8 M nodes, 168 M edges | 1.34 |

**Table 3: Benchmark information: source implementations, inputs, and execution time on a single-core system.**

## 5 EXPERIMENTAL METHODOLOGY

**Modeled system:** We use a cycle-accurate, event-driven simulator based on Pin [40, 47] to model FRACTAL systems of up to 256 cores, as shown in Fig. 8, with parameters in Table 2. Swarm parameters (task and commit queue sizes, etc.) match those from prior work [35, 36, 37]. We use detailed core, cache, network, and main memory models, and faithfully simulate all speculation overheads (e.g., running misspeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). Our 256-core configuration is similar to the Kalray MPPA [22]. We also simulate smaller systems with square meshes ($K \times K$ tiles for $K \leq 8$). 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.

**Benchmarks:** Table 3 reports the benchmarks we evaluate. Benchmarks have 1-core run-times of about 1 B cycles or longer. We

| | Perf. vs serial @ 1-core | | Avg task length (cycles) | | Nesting type |
|---|---|---|---|---|---|
| | `flat` | `fractal` | `flat` | `fractal` | |
| **maxflow** | 0.92× | 0.68× | 3260 | 373 | unord ↪ ord-32b |
| **labyrinth** | 1× | 0.62× | 16 M | 220 | unord ↪ ord-32b |
| **bayes** | 1× | 1.11× | 1.8 M | 3590 | unord ↪ unord |
| **silo** | 1.14× | 1.10× | 80 K | 3420 | unord ↪ ord-32b |
| **mis** | 0.79× | 0.26× | 162 | 115 | unord ↪ unord |
| **color** | 1.06× | 0.80× | 633 | 96 | ord-32b ↪ ord-32b |
| **msf** | 3.1× | 1.73× | 113 | 49 | ord-64b ↪ unord |

**Table 4: Benchmarks with parallel nesting: performance of 1-core `flat`/`fractal` vs tuned serial versions (higher is better), average task lengths, and nesting semantics.**

use three existing Swarm benchmarks [35, 36], which we adapt to Fractal; Fractal implementations of the eight STAMP benchmarks [42]; and two new Fractal benchmarks: `maxflow`, adapted from `prsn` [8], and `mis`, adapted from PBBS [54].

Benchmarks adapted from Swarm use their same inputs [35, 36]. `msf` includes an optimization to filter out non-spanning edges efficiently [9]. This optimization improves absolute performance but reduces the amount of highly parallel work, so `msf` has lower scalability than the unoptimized Swarm version [36].

STAMP benchmarks use inputs between the recommended "+" and "++" sizes, to achieve a run-time large enough to evaluate 256-core systems, yet small enough to be simulated in reasonable time.

`maxflow` uses rmf-wide [29], one of the harder graph families from the DIMACS maxflow challenge [8]. `mis` uses an R-MAT graph [15], which has a power-law distribution.

We fast-forward each benchmark to the start of its parallel region (skipping initialization), and report results for the full parallel region. On all benchmarks except `bayes`, we perform enough runs to achieve 95% confidence intervals $\leq$ 1%. `bayes` is highly non-deterministic, so we report its average results with 95% confidence intervals over 50 runs.

## 6 EVALUATION

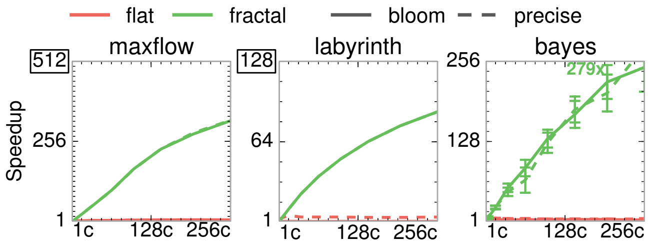
We now analyze the benefits of Fractal in depth. As in Sec. 2, we begin with applications where Fractal uncovers abundant fine-grain parallelism through nesting. We then discuss Fractal's benefits from avoiding over-serialization. Finally, we characterize the performance overheads of zooming to support deeper nesting.

### 6.1 Fractal uncovers abundant parallelism

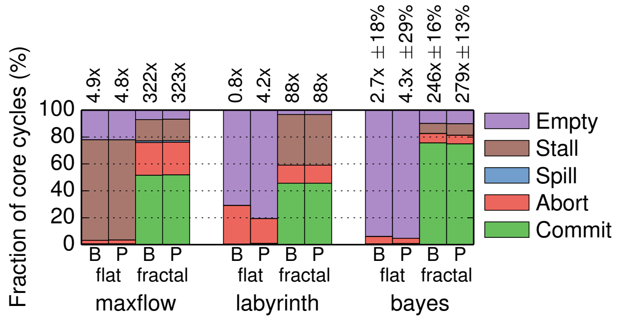
Fractal's support for nested parallelism greatly benefits three benchmarks: `maxflow`, as well as `labyrinth` and `bayes`, the two least scalable benchmarks from STAMP.

**maxflow**, as discussed in Sec. 2.1, is limited by long global-relabel tasks. Our `fractal` version performs the breadth-first search nested within each global relabel in parallel.

**labyrinth** finds non-overlapping paths between pairs of *(start, end)* cells on a 3D grid. Each transaction operates on one pair: it finds the shortest path on the grid and claims the cells on the path for itself. In the STAMP implementation, each transaction performs this shortest-path search sequentially. Our `fractal` version runs the shortest-path search nested within each transaction in parallel, using an ordered subdomain.



**(a) Speedup from 1 to 256 cores relative to 1-core `flat`.**



**(b) Breakdown of core cycles at 256 cores, with speedups on top.**

**Figure 14: Performance of `flat` and `fractal` versions of applications with abundant nested parallelism, using Bloom filter–based or Precise conflict detection.**

**bayes** learns the structure of a Bayesian network, a DAG where nodes denote random variables and edges denote conditional dependencies among variables. `bayes` spends most time deciding whether to insert, remove, or reverse network edges. Evaluating each decision requires performing many queries to an ADTree data structure, which efficiently represents probability estimates. In the STAMP implementation, each transaction evaluates and applies an insert/remove/reverse decision. Since the ADTree queries performed depend on the structure of the network, transactions serialize often. Our `fractal` version runs ADTree queries nested within each transaction in parallel, using an unordered subdomain.

Table 4 compares the 1-core performance and average task lengths of `flat` and `fractal` versions. `flat` versions of these benchmarks have long, unordered transactions (up to 16 M cycles). `fractal` versions have much smaller tasks (up to 3590 cycles on average in `bayes`). These short tasks hurt serial performance (by up to 38% in `labyrinth`), but expose plentiful *intra-domain parallelism* (e.g., a parallel breadth-first search), yielding great scalability.

Beyond limiting parallelism, the long transactions of `flat` versions have large read/write sets that often overflow Fractal's Bloom filters, causing false-positive aborts. Therefore, we also present results under an idealized, *precise* conflict detection scheme that does not incur false positives. High false positive rates are not specific to Fractal—prior HTMs used similarly-sized Bloom filters [14, 43, 53, 65].

Fig. 14a shows the performance of the `flat` and `fractal` versions when scaling from 1- to 256-core systems. All speedups reported are over the 1-core `flat` version. Solid lines show speedups when using Bloom filters, while dashed ones show speedups under precise conflict detection. `flat` versions scale poorly, especially when using Bloom filters: the maximum speedups across all system sizes range from 1.0× (`labyrinth` at 1 core) to 4.9× (`maxflow`). By contrast, `fractal` versions scale much better, from 88× (`labyrinth`) to 322× (`maxflow`).[3]

Fig. 14b gives more insight into these differences by showing the percentage of cycles that cores spend on different activities: *(i)* running tasks that are ultimately committed, *(ii)* running tasks that are later aborted, *(iii)* spilling tasks from the hardware task queues, *(iv)* stalled on a full task or commit queue, or *(v)* stalled due to lack of tasks. Each group of bars shows results for a different application at 256 cores.

Fig. 14b shows that `flat` versions suffer from lack of work caused by insufficient parallelism, and stalls caused by long tasks that eventually become the earliest active task and prevent others from committing. Moreover, most of the work performed by `flat` versions is aborted as tasks have large read/write sets and frequently conflict. `labyrinth-flat` and `bayes-flat` also suffer frequent false-positive aborts that hurt performance with Bloom filter conflict detection. Although precise conflict detection helps `labyrinth-flat` and `bayes-flat`, both benchmarks still scale poorly (to 4.3× and 6.8×, respectively) due to insufficient parallelism.

By contrast, `fractal` versions spend most cycles executing useful work, and aborted cycles are relatively small, from 7% (`bayes`) to 24% (`maxflow`). `fractal` versions perform just as well with Bloom filters as with precise conflict detection. These results shows that exploiting fine-grain nested speculative parallelism is an effective way to scale challenging applications.

## 6.2 Fractal avoids over-serialization

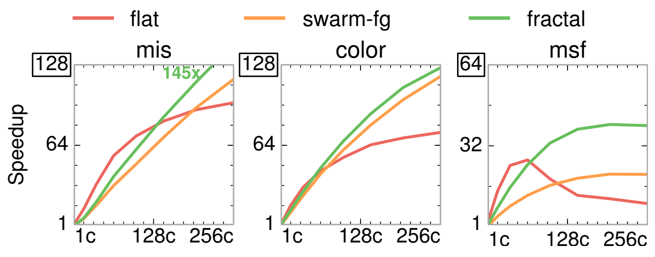
FRACTAL's support for nested parallelism avoids over-serialization on four benchmarks: `silo`, `mis`, `color`, and `msf`. Swarm can exploit nested parallelism in these benchmarks by imposing a total order among coarse-grain operations or groups of tasks (Sec. 2.3). Sec. 2 showed that this has a negligible effect on `silo`, so we focus on the other three applications.

`mis`, `color`, and `msf` are graph-processing applications. Their `flat` versions perform operations on multiple graph nodes that can be parallelized but must remain atomic—e.g., in `mis`, adding a node to the independent set and excluding its neighbors (Sec. 2.3). `mis-flat` is unordered, while `color-flat` and `msf-flat` visit nodes in a *partial* order (e.g., `color` visits larger-degree nodes first). Our `fractal` versions use one subdomain per coarse-grain operation to exploit this nested parallelism (Table 4). The `swarm-fg` versions of these benchmarks use the same fine-grain tasks as `fractal` but use a unique timestamp or timestamp range per coarse-grain operation to guarantee atomicity, imposing a *fixed* order among coarse-grain operations.

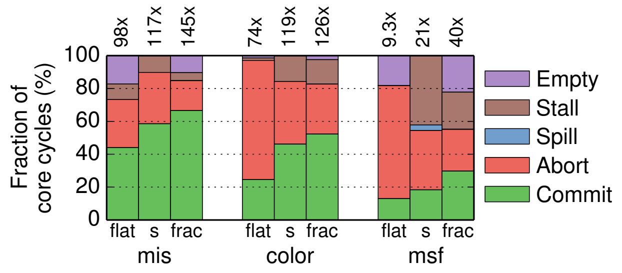
Fig. 15 shows the scalability and cycle breakdowns for these benchmarks. `flat` versions achieve the lowest speedups, from 26× (`msf` at 64 cores) to 98× (`mis`). Fig. 15b shows that they are dominated by aborts, which take up to 73% of cycles in `color-flat`, and empty cycles caused by insufficient parallelism in `msf` and `mis`. In `msf-flat`, frequent aborts hurt performance beyond 64 cores.

By contrast, `fractal` versions achieve the highest performance, from 40× (`msf`) to 145× (`mis`). At 256 cores, the majority of time

[3] Note that systems with more tiles have higher cache and queue capacities, which sometimes cause superlinear speedups (Sec. 5).



**(a) Speedup from 1 to 256 cores relative to 1-core `flat`.**



**(b) Breakdown of core cycles at 256 cores, with speedups on top.**

**Figure 15: Performance of `flat`, `swarm-fg`, and `fractal` versions of applications where Swarm extracts nested parallelism through strict ordering, but FRACTAL outperforms it by avoiding undue serialization.**

is spent on committed work, although aborts are still noticeable (up to 30% of cycles in `color`). While `fractal` versions perform better at 256 cores, their tiny tasks impose higher overheads, so they underperform `flat` on small core counts. This is most apparent in `msf`, where `fractal` tasks are just 49 cycles on average (Table 4).

Finally, `swarm-fg` versions follow the same scaling trends as `fractal` ones, but over-serialization makes them 6% (`color`), 24% (`mis`), and 93% (`msf`) slower. Fig. 15b shows that these slowdowns primarily stem from more frequent aborts. This is because in `swarm-fg` versions, conflict resolution priority is static (determined by timestamps), while in `fractal` versions, it is based on the dynamic execution order (determined by tiebreakers). In summary, these results show that FRACTAL makes fine-grain parallelism more attractive by avoiding needless order constraints.

## 6.3 Zooming overheads

Although our FRACTAL implementation supports unbounded nesting (Sec. 4.3), two nesting levels suffice for all the benchmarks we evaluate. Larger programs should require deeper nesting. Therefore, we use a microbenchmark to characterize the overheads of FRACTAL's zooming technique.

Our microbenchmark stresses FRACTAL by creating many nested domains that contain few tasks each. Specifically, it generates a depth-8 tree of nested domains with fanout $F$. All tasks perform a small, fixed amount of work (1500 cycles). Non-leaf tasks then create an unordered subdomain and enqueue $F$ children into it. We sweep both the fanout ($F = 4$ to 12) and the maximum number of concurrent levels $D$ in FRACTAL, from 2 (64-bit fractal VTs) to 8 (256-bit fractal VTs). At $D = 8$, the system does not perform any zooming. Our default hardware configuration supports up to 4 concurrent levels.

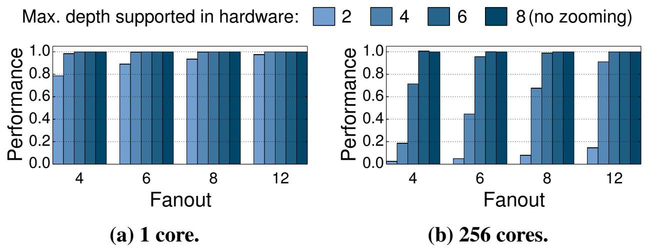


**Figure 16: Characterization of zooming overheads.**

Fig. 16a reports performance on a 1-core system. Each group of bars shows results for a single fanout, and bars within a group show how performance changes as the maximum concurrent levels $D$ grows from 2 to 8. Performance is relative to the $D = 8$, no-zooming system. Using a 1-core system lets us focus on the overheads of zooming without factoring in limited parallelism. Larger fanouts and concurrent levels increase the amount of work executed between zooming operations, reducing overheads. Nonetheless, overheads are modest even for $F = 4$ and $D = 2$ (21% slowdown).

Fig. 16b reports performance on a 256-core system. Supporting a limited number of levels reduces parallelism, especially with small fanouts, which hurts performance. Nonetheless, as long as $F \geq 8$, supporting at least four levels keeps overheads small.

All of our applications have much higher parallelism than 8 concurrent tasks in at least one of their two nesting levels, and often in both. Therefore, on applications with deeper nesting, zooming should not limit performance in most cases. However, these are carefully coded applications that avoid unnecessary nesting. Nesting could be overused (e.g., increasing the nesting depth at every intermediate step of a divide-and-conquer algorithm), which would limit parallelism. To avoid this, a compiler pass may be able to safely flatten unnecessary nesting levels. We leave this to future work.

## 6.4 Discussion

We considered 18 benchmarks to evaluate FRACTAL: all eight from Swarm [35, 36], all eight from STAMP [42], as well as `maxflow` and `mis`. We looked for opportunities to exploit nested parallelism, focusing on benchmarks with limited speedups. In summary, FRACTAL benefits 7 out of these 18 benchmarks. We did not find opportunities to exploit nested parallelism in the five Swarm benchmarks not presented here (`bfs`, `sssp`, `astar`, `des`, and `nocsim`). These benchmarks already use fine-grain tasks and scale well to 256 cores.

Fig. 17 shows how each STAMP benchmark scales when using different FRACTAL features. All speedups reported are over the 1-core TM version. The TM lines show the performance of the original STAMP transactions ported to Swarm tasks. Three applications (`intruder`, `labyrinth`, and `bayes`) barely scale, while two (`yada` and `kmeans`) scale well at small core counts but suffer on larger systems. By contrast, FRACTAL's features make all STAMP applications scale, although speedups are not only due to nesting. First, the TM versions of `intruder` and `yada` use software task queues that limit their scalability. Refactoring them to use Swarm/FRACTAL hardware task queues [36] makes them scale. Second, spatial hints [35] improves `genome` and makes `kmeans` scale. Finally, as we saw in Sec. 6.1, FRACTAL's support for nesting makes

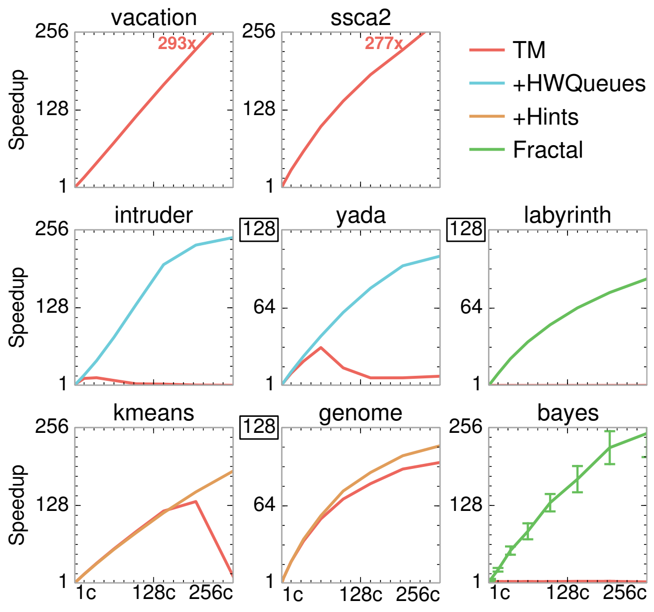


**Figure 17: Different FRACTAL features make all STAMP applications scale well to 256 cores.**

`labyrinth` and `bayes` scale. Therefore, FRACTAL is the first architecture that scales the full STAMP suite to hundreds of cores, achieving a gmean speedup of 177× at 256 cores.

## 7 RELATED WORK

### 7.1 Nesting in transactional memory

**Serial nesting:** Most HTMs support *serial* execution of nested transactions, which makes transactional code easy to compose but forgoes intra-transaction parallelism. Nesting can be trivially supported by ignoring the boundaries of all nested transactions, treating them as part of the top-level one. Some HTMs exploit nesting to implement *partial aborts* [44]: they track the speculative state of a nested transaction separately while it executes, so conflicts that occur while the nested transaction runs do not abort the top-level one.

Even with partial aborts, HTMs ultimately merge nested speculative state into the top-level transaction, resulting in large atomic regions that are hard to support in hardware [2, 12, 19, 20] and make conflicts more likely.

Prior work has explored relaxed nesting semantics, like open nesting [41, 44, 46] and early release [55], which relax isolation to improve performance. FRACTAL is orthogonal to these techniques and could be extended to support them, but we do not see the need on the applications we study.

**Parallel nesting:** Some TM systems support running nested transactions in parallel [45]: a transaction can launch multiple nested transactions and wait for them to finish. Nested transactions may run in parallel and can observe updates from their parent transaction. As in serial nesting, when a nested transaction finishes, its speculative state is merged with its parent's. When all nested transactions finish, the parent transaction resumes execution.

Most of this work has been in software TM (STM) implementations [1, 5, 24, 63], but these suffer from even higher overheads than flat STMs. Vachharajani [62, Ch. 7] and FaNTM [6] introduce hardware support to reduce parallel nesting overheads. Even with hardware support, parallel-nesting HTMs yield limited gains—e.g., FaNTM is often slower than a flat HTM, and moderately outperforms it (by up to 40%) only on a microbenchmark.

Parallel-nesting TMs suffer from three main problems. First, nested transactions merge their speculative state with their parent's, and only the coarse, top-level transaction can commit. This results in large atomic blocks that are as expensive to track and as prone to abort as large serial transactions. By contrast, FRACTAL performs fine-grain speculation, at the level of individual tasks. It never merges the speculative state of tasks, and relies on ordering tasks to guarantee the atomicity of nested domains.

Second, because the parent transaction waits for its nested transactions to finish, there is a cyclic dependence between the parent and its nested transactions. This introduces many subtle problems, including data races with the parent, deadlock, and livelock [6]. Workarounds for these issues are complex and sacrifice performance (e.g., a nested transaction eventually aborts all its ancestors for livelock avoidance [6]). By contrast, all dependences in FRACTAL are acyclic, from parents to children, which avoids these issues. FRACTAL supports the fork-join semantics of parallel-nesting TMs by having nested transactions enqueue their parent's continuation.

Finally, parallel-nesting TMs do not support ordered speculative parallelism. By contrast, FRACTAL supports arbitrary nesting of ordered and unordered parallelism, which accelerates a broader range of applications.

### 7.2 Thread-level speculation

Thread-level speculation (TLS) schemes [28, 31, 50, 56, 58] ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Prior TLS systems scale poorly beyond few cores, cannot support large speculation windows, and are less general than Swarm's timestamp-ordered execution model [36].

A few TLS systems use timestamps internally, but do not let programs control them [32, 50, 57]. Renau et al. [50] use timestamps to allow out-of-order task spawn. Each task carries a timestamp range, and splits it in half when it spawns a successor. This approach could be adapted to support the order constraints required by nesting. However, while this technique works well at the scale it was evaluated (4 speculative tasks), it would require an impractical number of timestamp bits at the scale we consider (4096 speculative tasks). Moreover, this technique would cause over-serialization and does not support exposing timestamps to programs.

### 7.3 Nesting with non-speculative parallelism

Nesting is supported by most parallel programming languages, such as OpenMP [26]. In many languages, such as NESL [10], Cilk [27], and X10 [16], nesting is the natural way to express parallelism. Supporting nested parallelism in these non-speculative systems is easy because parallel tasks have no atomicity requirements: they either operate on disjoint data or use explicit synchronization, such as locks [17] or dataflow annotations [25], to avoid data races. Though nested non-speculative parallelism is often sufficient, many algorithms need speculation to be parallelized efficiently [48]. By making nested speculative parallelism practical, FRACTAL brings the benefits of composability and fine-grain parallelism to a broader set of programs.

## 8 CONCLUSION

We have presented FRACTAL, a new execution model for fine-grain nested speculative parallelism. FRACTAL lets programmers compose ordered and unordered algorithms without undue serialization. Our FRACTAL implementation builds on the Swarm architecture and relies on a dynamically chosen task order to perform fine-grain speculation, operating at the level of individual tasks. Our implementation sidesteps the scalability issues of parallel-nesting HTMs and requires simple hardware. We have shown that FRACTAL can parallelize a broader range of applications than prior work, and outperforms prior speculative architectures by up to 88×.

## ACKNOWLEDGMENTS

We sincerely thank Nathan Beckmann, Nosayba El-Sayed, Harshad Kasture, Po-An Tsai, Guowei Zhang, Anurag Mukkara, Yee Ling Gan, and the anonymous reviewers for their helpful feedback. Niklas Baumstark and Julian Shun graciously shared their serial and parallel implementations of `maxflow`.

This work was supported in part by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grants CAREER-1452994 and CCF-1318384. Mark C. Jeffrey was partially supported by an NSERC postgraduate scholarship; Hyun Ryong Lee was partially supported by a Kwanjeong Educational Foundation scholarship; and Victor A. Ying was supported by an MIT EECS Edwin S. Webster Graduate Fellowship.

## REFERENCES

[1] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. 2008. Nested parallelism in transactional memory. In *Proc. PPoPP*.
[2] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded transactional memory. In *Proc. HPCA-11*.
[3] Richard J. Anderson and João C. Setubal. 1992. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proc. SPAA*.
[4] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2012. *10th DIMACS Implementation Challenge Workshop*.
[5] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proc. SPAA*.
[6] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Making nested parallel transactions practical using lightweight hardware support. In *Proc. ICS'10*.
[7] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. 2007. The OpenTM transactional application programming interface. In *Proc. PACT-16*.
[8] Niklas Baumstark, Guy Blelloch, and Julian Shun. 2015. Efficient implementation of a synchronous parallel push-relabel algorithm. In *Proc. ESA*.
[9] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. PPoPP*.
[10] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. In *Proc. PPoPP*.
[11] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M.K. Martin. 2007. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proc. ISCA-34*.
[12] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. 2008. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proc. ISCA-35*.
[13] J. Lawrence Carter and Mark Wegman. 1977. Universal classes of hash functions (Extended abstract). In *Proc. STOC-9*.
[14] Luis Ceze, James Tuck, Josep Torrellas, and Călin Caşcaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. ISCA-33*.

[15] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proc. SDM*.

[16] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA-20*.

[17] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proc. SPAA*.

[18] Boris V. Cherkassky and Andrew V. Goldberg. 1997. On implementing the push-relabel method for the maximum flow problem. *Algorithmica* 19, 4 (1997).

[19] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. 2006. Unbounded page-based transactional memory. In *Proc. ASPLOS-XII*.

[20] JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. 2006. Tradeoffs in transactional memory virtualization. In *Proc. ASPLOS-XII*.

[21] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM TOMS* 38, 1 (2011).

[22] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, and Thierry Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *Proc. HPEC*.

[23] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic shared memory multiprocessing. In *Proc. ASPLOS-XIV*.

[24] Nuno Diegues and João Cachopo. 2013. Practical parallel nesting for software transactional memory. In *Proc. DISC*.

[25] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21, 02 (2011).

[26] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. Evaluation of OpenMP task scheduling strategies. In *4th Intl. Workshop in OpenMP*.

[27] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proc. PLDI*.

[28] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. 2003. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. In *Proc. HPCA-9*.

[29] Donald Goldfarb and Michael D. Grigoriadis. 1988. A computational comparison of the dinic and network simplex methods for maximum flow. *Annals of Operations Research* 13, 1 (1988).

[30] Daniel C. Halbert and Peter B. Kessler. 1980. Windows of overlapping register frames. CS 292R Final Report, UC Berkeley. (1980).

[31] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proc. ASPLOS-VIII*.

[32] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *Proc. ISCA-31*.

[33] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proc. SPAA*.

[34] David R. Jefferson. 1985. Virtual time. *ACM TOPLAS* 7, 3 (1985).

[35] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-centric execution of speculative parallel programs. In *Proc. MICRO-49*.

[36] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. MICRO-48*.

[37] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2016. Unlocking ordered parallelism with the Swarm architecture. *IEEE Micro* 36, 3 (2016).

[38] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. ASPLOS-X*.

[39] Jure Leskovec and Andrej Krevl. 2014. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data. (2014).

[40] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*.

[41] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. 2006. Architectural semantics for practical transactional memory. In *Proc. ISCA-33*.

[42] Chí Cao Minh, Jaewoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. IISWC*.

[43] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. ISCA-34*.

[44] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. 2006. Supporting nested transactional memory in LogTM. In *Proc. ASPLOS-XII*.

[45] J. Eliot B. Moss and Antony L. Hosking. 2006. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming* 63, 2 (2006).

[46] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. 2007. Open nesting in software transactional memory. In *Proc. PPoPP*.

[47] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News* 33, 5 (2005).

[48] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proc. PLDI*.

[49] Naresh Rapolu, Karthik Kambatla, Suresh Jagannathan, and Ananth Grama. 2011. TransMR: Data-centric programming beyond data parallelism. In *HotCloud*.

[50] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. 2005. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *Proc. ICS'05*.

[51] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier?. In *Proc. PPoPP*.

[52] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. ISCA-40*.

[53] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. 2007. Implementing signatures for transactional memory. In *Proc. MICRO-40*.

[54] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: The Problem Based Benchmark Suite. In *Proc. SPAA*.

[55] Travis Skare and Christos Kozyrakis. 2006. Early release: Friend or foe?. In *Proc. WTW*.

[56] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proc. ISCA-22*.

[57] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proc. ISCA-27*.

[58] J. Gregory Steffan and Todd C. Mowry. 1998. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. HPCA-4*.

[59] Gerald Jay Sussman and Guy L. Steele Jr. 1998. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation* 11, 4 (1998).

[60] Andrew S. Tanenbaum and David J. Wetherall. 2010. *Computer networks* (5th ed.).

[61] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proc. SOSP-24*.

[62] Neil Vachharajani. 2008. *Intelligent speculation for pipelined multithreading*. Ph.D. Dissertation. Princeton University.

[63] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. 2009. NePalTM: Design and implementation of nested parallelism for transactional memory systems. In *ECOOP*.

[64] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. 2007. On-chip interconnection architecture of the Tile Processor. *IEEE Micro* 27, 5 (2007).

[65] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. HPCA-13*.