Data-Centric Execution of Speculative Parallel Programs

Mark C. Jeffrey* Suvinay Subramanian* Maleen Abeydeera* Joel Emer† Daniel Sanchez*
*Massachusetts Institute of Technology †NVIDIA / MIT
{mcj, suvinay, maleen, emer, sanchez}@csail.mit.edu

***Abstract*—Multicore systems must exploit locality to scale, scheduling tasks to minimize data movement. While locality-aware parallelism is well studied in non-speculative systems, it has received little attention in speculative systems (e.g., HTM or TLS), which hinders their scalability.**

We present *spatial hints*, a technique that leverages program knowledge to reveal and exploit locality in speculative parallel programs. A hint is an abstract integer, given when a speculative task is created, that denotes the data that the task is likely to access. We show it is easy to modify programs to convey locality through hints. We design simple hardware techniques that allow a state-of-the-art, tiled speculative architecture to exploit hints by: *(i)* running tasks likely to access the same data on the same tile, *(ii)* serializing tasks likely to conflict, and *(iii)* balancing tasks across tiles in a locality-aware fashion. We also show that programs can often be restructured to make hints more effective.

Together, these techniques make speculative parallelism practical on large-scale systems: at 256 cores, hints achieve near-linear scalability on nine challenging applications, improving performance over hint-oblivious scheduling by 3.3× gmean and by up to 16×. Hints also make speculation far more efficient, reducing wasted work by 6.4× and traffic by 3.5× on average.

I. Introduction

Speculative parallelization, e.g. through thread-level speculation (TLS) or hardware transactional memory (HTM), has two major benefits over non-speculative parallelism: it uncovers abundant parallelism in many challenging applications [28, 37] and simplifies parallel programming [59]. However, even with scalable versioning and conflict detection techniques [14, 37, 56, 67], speculative systems scale poorly beyond a few tens of cores. A key reason is that these systems *do not exploit much of the locality available in speculative programs*.

To scale, parallelism must not come at the expense of locality: tasks should be run close to the data they access to avoid global communication and use caches effectively. The need for locality-aware parallelism is well understood in non-speculative systems, where abundant prior work has developed programming models to convey locality [4, 69, 74], and runtimes and schedulers to exploit it [2, 10, 15, 33, 50, 65, 76].

However, most prior work in speculative parallelization has ignored the need for locality-aware parallelism: in TLS, speculative tasks are executed by available cores without regard for locality [27, 57, 66]; and conventional HTM programs are structured as threads that execute a fixed sequence of transactions. Prior work has observed that it is beneficial to structure transactional code into tasks instead, and has proposed transactional task schedulers that *limit concurrency* to reduce aborts under high contention [5, 6, 8, 9, 19, 21, 35, 61, 77]. Limiting concurrency suffices for small systems, but scaling to hundreds of cores also requires solving the *spatial mapping* problem: speculative tasks must be mapped across the system to minimize data movement.

To our knowledge, no prior work has studied the spatial mapping problem for speculative architectures. This may be because, at first glance, spatial mapping and speculation seem to be at odds: achieving a good spatial mapping requires knowing the data accessed by each task, but the key advantage of speculation is precisely that one need not know the data accessed by each task. However, we find that *there is a wide gray area*: in many applications, *most* of the data accessed is known at runtime when the task is created. Thus, there is ample information to achieve high-quality spatial task mappings. Beyond reducing data movement, high-quality mappings also enhance parallelism by making most conflicts local.

To exploit this insight, we present *spatial hints*, a technique that uses program knowledge to achieve high-quality task mappings (Sec. III). A hint is an abstract integer, given at runtime when a task is created, that denotes the data that the task is likely to access. We show it is easy to modify programs to convey locality through hints. We enhance a state-of-the-art tiled speculative architecture, Swarm [37, 38], to exploit hints by sending tasks with the same hint to the same tile and running them serially.

We then analyze how task structure affects the effectiveness of hints (Sec. V). We find that fine-grain tasks access less data, and more of that data is known at task creation time, making hints more effective. Although programs with fine-grain tasks perform more work and stress scheduler overheads, hints make fine-grain tasks a good tradeoff by reducing memory stalls and conflicts further. We show that certain programs can be easily restructured to use finer-grain tasks (Sec. V), improving performance by up to 2.7×.

Finally, while hints improve locality and reduce conflicts, they can also cause load imbalance. We thus design a load balancer that leverages hints to redistribute tasks across tiles in a locality-aware fashion (Sec. VI). Unlike non-speculative load balancers, the signals to detect imbalance are different with speculation (e.g., tiles do not run out of tasks, but run tasks that are likely to abort), requiring a different approach. Our load balancer improves performance by up to a further 27%.

In summary, we present four novel contributions:

978-1-5090-3508-3/16/$31.00

- Spatial hints, a technique that conveys program knowledge to achieve high-quality spatial task mappings.
- Simple hardware mechanisms to exploit hints by sending tasks likely to access the same data to the same place and running them serially.
- An analysis of the relationship between task granularity and locality, showing that programs can often be restructured to make hints more effective.
- A novel data-centric load-balancer that leverages hints to redistribute tasks without hurting locality.

Together, these techniques make speculative parallelism practical on large-scale systems: at 256 cores, hints achieve near-linear scalability on nine challenging applications, outperform the baseline Swarm system by 3.3× gmean and by up to 16×, and outperform a work-stealing scheduler by a wider margin. Hints also make speculation far more efficient, reducing wasted work by 6.4× and network traffic by 3.5× on average.

II. Background and Motivation

We demonstrate the benefits of spatial task mapping on Swarm [37, 38], a recent architecture for speculative parallelization. We choose Swarm as a baseline for two key reasons. First, Swarm's task-based execution model is general: it supports ordered and unordered parallelism, subsuming both TLS and TM, and allows more ordered programs to be expressed than TLS. This allows us to test our techniques with a broader range of speculative programs than alternative baselines. Second, Swarm focuses on efficiently supporting fine-grain tasks, and includes hardware support for task creation and queuing. This allows us to study the interplay between task granularity and spatial hints more effectively than alternative baselines with software schedulers, which are limited to coarse-grain tasks.

We first present Swarm's main features (please see prior work [37, 38] for details). We then motivate the need for spatial task mapping through a simple example.

A. Swarm Execution Model

Swarm programs consist of timestamped tasks. Each task may access arbitrary data, and can create child tasks with any timestamp greater than or equal to its own. Swarm guarantees that tasks appear to run in timestamp order. If multiple tasks have equal timestamp, Swarm chooses an order among them.

Swarm exposes its execution model through a simple API. Listing 1 illustrates this API by showing the Swarm implementation of `des`, a discrete event simulator for digital circuits adapted from Galois [31, 54]. Note that parallelism is implicit—there is no synchronization or thread management.

Each task runs a function that takes a timestamp and an arbitrary number of additional arguments. Listing 1 defines one task function, `desTask`, which simulates a signal toggling at a gate input. Tasks can create child tasks by calling `swarm::enqueue` with the appropriate task function, timestamp, and arguments. In our example, if an input toggle causes the gate output to toggle, `desTask` enqueues child tasks for all the gates connected to this output. Finally, a program invokes Swarm by enqueuing some initial tasks with `swarm::enqueue` and calling `swarm::run`, which returns control when all tasks finish. For example, Listing 1 enqueues a task for each input waveform, then starts the simulation.

```
void desTask(Timestamp ts, GateInput* input) {
  Gate* g = input->gate();
  bool toggledOutput = g.simulateToggle(input);
  if (toggledOutput) {
    // Toggle all inputs connected to this gate
    for (GateInput* i : g->connectedInputs())
      swarm::enqueue(desTask, ts + delay(g, i), i);
  }
}

void main() {
  [...]  // Set up gates and initial values
  // Enqueue events for input waveforms
  for (GateInput* i : externalInputs)
    swarm::enqueue(inputWaveformTask, 0, i);
  swarm::run();  // Start simulation
}
```

Listing 1. Swarm implementation of discrete event simulation for digital circuits.

Swarm's execution model supports both TLS-style ordered speculation by choosing timestamps that reflect the serial order as in prior work [58], and TM-style unordered speculation by using the same timestamp for all tasks. Moreover, Swarm's execution model generalizes TLS by *decoupling task creation and execution orders*: whereas in prior TLS schemes tasks could only spawn speculative tasks that were immediate successors [27, 28, 58, 66, 67], Swarm tasks can create child tasks with any timestamp equal or higher than their own. This allows programs to convey new work to hardware as soon as it is discovered instead of in the order it needs to run, exposing a large amount of parallelism for ordered irregular applications (typical in e.g., graph analytics, simulation, and databases [37]). While in our prior work we used Swarm for ordered speculation only [37], here we study both ordered and unordered speculative programs.

B. Swarm Microarchitecture

The Swarm microarchitecture uncovers parallelism by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest active task. Swarm introduces modest changes to a tiled, cache-coherent multicore, shown in Fig. 1. Each tile has a group of simple cores, each with its own private L1 caches. 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 efficiently supports fine-grain tasks and a large speculation window through four main mechanisms: low-overhead hardware task management, large task queues, scalable speculation mechanisms, and high-throughput ordered commits.

Hardware task management: Each tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Swarm executes every task except the earliest active task speculatively. To uncover enough parallelism, task units can dispatch any available task to cores,

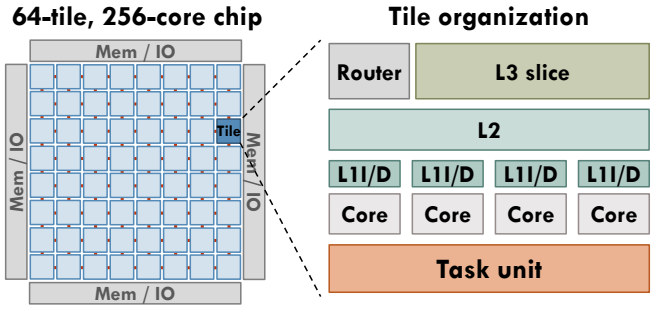
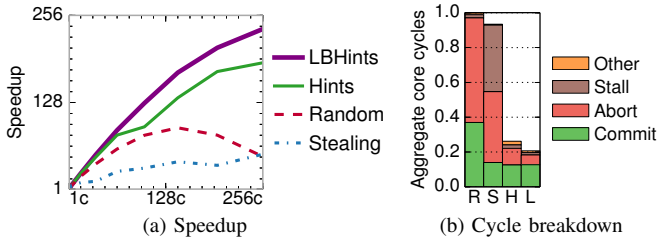


Fig. 1. Swarm system and tile configuration.



(a) Speedup (b) Cycle breakdown

Fig. 2. Performance of Random, Stealing, Hints, and LBHints schedulers on des: (a) speedup relative to 1-core Swarm, and (b) breakdown of total core cycles at 256 cores, relative to Random.

no matter how distant in program order. A task can run even if its parent is still speculative.

Each task is represented by a task descriptor that contains its function pointer, 64-bit timestamp, and arguments. Cores dequeue tasks for execution in timestamp order from the local task unit. Successful dequeues initiate speculative execution at the task's function pointer and make the task's timestamp and arguments available in registers. A core stalls if there is no task to dequeue. Tasks create child tasks and enqueue them to other tiles.

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

Task and commit queues support tens of speculative tasks per core (e.g., 64 task queue entries and 16 commit queue entries per core) to implement a large window of speculation (e.g., 16 thousand tasks in the 256-core chip in Fig. 1). Nevertheless, because programs can enqueue tasks with arbitrary timestamps, task and commit queues can fill up. This requires some simple actions to ensure correct behavior. Specifically, tasks that have not started execution and whose parent has committed are *spilled* to memory to free task queue entries. For all other tasks, queue resource exhaustion is handled by either stalling the enqueuer or aborting higher-timestamp tasks to free space [37].

Scalable speculation: Swarm leverages previously proposed speculation mechanisms and enhances them to support a large number of speculative tasks. Swarm uses eager (undo-log-based) version management and eager conflict detection using Bloom filters, similar to LogTM-SE [75]. Swarm forwards still-speculative data read by a later task; on an abort, Swarm aborts only descendants and data-dependent tasks. Swarm features several techniques to substantially reduce the number of conflict checks and their cost [37].

High-throughput ordered commits: Finally, Swarm adapts the virtual time algorithm [36] to achieve high-throughput ordered commits. Tiles communicate with an arbiter periodically (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 achieves high commit rates, up to multiple tasks per cycle on average, which allows fine-grain ordered tasks, as short as a few tens of cycles.

These techniques let us study a broad range of speculative programs and harness the benefits of fine-grain tasks for locality-aware parallelism, but are otherwise orthogonal to the spatial mapping techniques we present in this paper.

C. Motivation for Spatial Task Mapping

To explore the impact of spatial task mapping, we compare the performance of the des benchmark from Listing 1 under different schedulers. We simulate systems of up to 256 cores as shown in Fig. 1 (see Sec. IV-A for methodology details). We compare four schedulers:

- Random is Swarm's default task mapping strategy. New tasks are sent to a random tile for load balance.
- Stealing is an *idealized work-stealing* scheduler, the most common scheduler in non-speculative programs [2, 10]. New tasks are enqueued to the local tile, and tiles that run out of tasks steal tasks from a victim tile. To evaluate stealing in the best possible light, we do not simulate any stealing overheads: out-of-work tiles instantaneously find the tile with the most idle tasks and steal the earliest-timestamp task. Work-stealing is sensitive to the stealing policies used (Sec. VII-B). We studied a wide range of policies, both in terms of victim tile selection (random, nearest-neighbor, most-loaded) and task selection within a victim tile (earliest-timestamp, random, latest-timestamp) and empirically found the selected policies to perform best overall for our benchmarks.
- Hints is our hint-based spatial task mapping scheme.
- LBHints is our hint-based load balancer.

In des, Hints maps each gate in the simulated circuit to a specific, statically chosen tile. New tasks are sent to the tile where the gate they operate on is mapped (Sec. III). LBHints enhances Hints by periodically remapping gates across tiles to equalize their load (Sec. VI).

Two factors make spatial mapping in des possible. First, each task operates on a single gate. Second, this gate is known at runtime when the task is created. As we will see later, good spatial mappings are possible even when these conditions are not completely met (i.e., tasks access multiple pieces of data, or some of the data they access is not known at task creation time). Also, note that even when we know all data accesses, speculation is still needed, as tasks can be created out of order and executed in the wrong order.

Fig. 2a compares the performance of different schemes on 1- to 256-core systems. Each line shows the speedup relative to a 1-core Swarm system (all schedulers are equivalent at 1

core). Stealing performs worst, scaling to 52× at 256 cores. Random peaks at 144 cores, scaling to 91×, and drops to 49× at 256 cores. Hints scales to 186×, and LBHints performs best, with a 236× speedup at 256 cores.

Fig. 2b yields further insights into these differences. The height of each bar in Fig. 2b is the sum of cycles spent by all cores, normalized to the cycles of Random (lower is better). Each bar shows the breakdown of cycles spent executing tasks that are ultimately committed, eventually aborted, cycles stalled on a full queue, and spent in other overheads. Most cycles are spent running committed tasks, aborted tasks, or in queue stalls, and trends are widely different across schemes.

Committed cycles mainly depend on locality: in the absence of conflicts, the only difference is memory stalls. Random has the highest committed cycles (most stalls), while Hints and LBHints have the lowest, as gates are held in nearby private caches. Stealing has slightly higher committed cycles, as it often keeps tasks for nearby gates in the same tile.

Differences in aborted cycles are higher. In `des`, conflict frequency depends highly on how closely tasks from different tiles follow timestamp order. Random and LBHints keep tiles running tasks with close-by timestamps. However, conflicts in LBHints are local, and thus much faster, and LBHints serializes tasks that operate on the same gate. For these reasons, LBHints spends the fewest cycles on aborts. Hints is less balanced, so it incurs more conflicts than LBHints. Finally, in Stealing, tiles run tasks widely out of order, as stealing from the most loaded tile is not a good strategy to maintain order in `des` (as we will see, this is a good strategy in other cases). This causes both significant aborts and queue stalls in Stealing, as commit queues fill up. These effects hinder Stealing's scalability.

Overall, these results show that hints can yield significant gains by reducing both aborts and data movement.

III. Spatial Task Mapping with Hints

We now present *spatial hints*, a general technique that leverages application-level knowledge to achieve high-quality task mappings. A hint is simply an abstract integer value, given at task creation time, that denotes the data likely to be accessed by a task. Hardware leverages hints to map tasks likely to access the same data to the same location.

We present the API and ISA extensions to support hints, describe the microarchitectural mechanisms to exploit hints, and show how to apply hints to a wide variety of benchmarks.

A. Hint API and ISA Extensions

We extend the `swarm::enqueue` function (Sec. II-A) with one field for the spatial hint:

```
swarm::enqueue(taskFn, timestamp, hint, args...)
```

This hint can take one of three values:

- A 64-bit integer value that conveys the data likely to be accessed. The programmer is free to choose what this integer represents (e.g., addresses, object ids, etc.). The only guideline is that tasks likely to access the same data should have the same hint.
- `NOHINT`, used when the programmer does not know what data will be accessed.
- `SAMEHINT`, which assigns the parent's hint to the child task.

Code enqueues tasks with an `enqueue_task` instruction that takes the function pointer, timestamp, and arguments in registers. We employ unused bits in the instruction opcode to represent whether the new task is tagged with an integer hint, `NOHINT`, or `SAMEHINT`. If tagged with an integer hint, we pass that value through another register.

B. Hardware Mechanisms

Hardware leverages hints in two ways:

1. Spatial task mapping: When a core creates a new task, the local task unit uses the hint to determine its destination tile. The task unit hashes the 64-bit hint down to a tile ID (e.g., 6 bits for 64 tiles), then sends the task descriptor to the selected tile. `SAMEHINT` tasks are queued to the local task queue, and `NOHINT` tasks are sent to a random tile.

2. Serializing conflicting tasks: Since two tasks with the same hint are likely to conflict, we enhance the task dispatch logic to avoid running them concurrently. Specifically, tasks carry a 16-bit hash of their hint throughout their lifetime. By default, the task unit selects the earliest-timestamp idle task for execution. Instead, we check whether that candidate task's hint hash matches one of the already-running tasks. If there is a match and the already-running task has an earlier timestamp, the task unit skips the candidate and tries the idle task with the next lowest timestamp.

Using 16-bit hashed hints instead of full hints requires less storage and simplifies the dispatch logic. Their lower resolution introduces a negligible false-positive match probability ($6 \cdot 10^{-5}$ with four cores per tile).

Overheads: These techniques add small overheads:

- 6- and 16-bit hash functions at each tile to compute the tile ID and hashed hint.
- An extra 16 bits per task descriptor. Descriptors are sent through the network (so hints add some traffic) and stored in task queues. In our chosen configuration, each tile's task queue requires 512 extra bytes.
- Four 16-bit comparators used during task dispatch.

C. Adding Hints to Benchmarks

We add hints to a diverse set of nine benchmarks. Table I summarizes their provenance, input sets, and the strategies used to assign hints to each benchmark.

Seven of our benchmarks are ordered:

- `bfs` finds the breadth-first tree of an arbitrary graph.
- `sssp` uses Dijkstra's algorithm to find the shortest-path tree of a weighted graph.
- `astar` uses the A$*$ pathfinding algorithm [29] to find the shortest route between two points in a road map.
- `color` uses the largest-degree-first heuristic [71] to assign distinct colors to adjacent graph vertices. This heuristic produces high-quality results and is thus most frequently used, but it is hard to parallelize.

TABLE I
BENCHMARK INFORMATION: SOURCE IMPLEMENTATIONS, INPUTS, RUN-TIMES ON A 1-CORE SWARM SYSTEM, 1-CORE SPEEDUPS OVER TUNED SERIAL IMPLEMENTATIONS, NUMBER OF TASK FUNCTIONS, AND HINT PATTERNS USED.

	Source	Input	Swarm 1-core Run-time	Swarm 1-core Perf vs serial	Task Funcs	Hint patterns
bfs	PBFS [43]	hugetric-00020 [7, 17]	3.59 Bcycles	−18%	1	Cache line of vertex
sssp	Galois [54]	East USA roads [1]	3.21 Bcycles	+33%	1	Cache line of vertex
astar	[37]	Germany roads [52]	1.97 Bcycles	+1%	1	Cache line of vertex
color	[30]	com-youtube [45]	1.65 Bcycles	+54%	3	Cache line of vertex
des	Galois [54]	csaArray32	1.92 Bcycles	+70%	8	Logic gate ID
nocsim	GARNET [3]	16x16 mesh, tornado traffic	22.37 Bcycles	+68%	10	Router ID
silo	[70]	TPC-C, 4 whs, 32 Ktxns	2.83 Bcycles	+16%	16	(Table ID, primary key)
genome	STAMP [48]	-g4096 -s48 -n1048576	2.30 Bcycles	+1%	10	Elem addr, map key, NO/SAMEHINT
kmeans	STAMP [48]	-m40 -n40 -i rnd-n16K-d24-c16	8.56 Bcycles	+2%	5	Cache line of point, cluster ID

- `des` is a simulator for digital circuits (Listing 1).
- `nocsim` is a detailed network-on-chip simulator derived from GARNET [3]. Each task simulates an event at a component of a router.
- `silo` is an in-memory OLTP database [70].

`bfs`, `sssp`, `astar`, `des`, and `silo` are from the original Swarm paper [37]; we develop `color` and `nocsim` by porting existing serial implementations. In either case, applications do not change the amount of work in the serial code. As shown in Table I, at 1 core, Swarm outperforms tuned serial versions in all cases except `bfs`, where 1-core Swarm is 18% slower.

We also port two unordered benchmarks from STAMP [48]:

- `genome` performs gene sequencing.
- `kmeans` implements K-means clustering.

We implement transactions with tasks of equal timestamp, so that they can commit in any order. As in prior work in transaction scheduling [5, 77] (Sec. VII), we break the original threaded code into tasks that can be scheduled asynchronously and generate children tasks as they find more work to do.

We observe that a few common *patterns* arise naturally when adding hints to these applications. We explain each of these patterns through a representative application.

Cache-line address: Our graph analytics applications (`bfs`, `sssp`, `astar`, and `color`) are vertex-centric [47]: each task operates on one vertex and visits its neighbors. For example, Listing 2 shows the single task function of `sssp`. Given the distance to the source of vertex `v`, the task visits each neighbor `n`; if the projected distance to `n` is reduced, `n`'s distance is updated and a new task created for `n`. Tasks appear to execute in timestamp order, i.e. the projected distance to the source.

```
void ssspTask(Timestamp pathDist, Vertex* v) {
  if (pathDist == v->distance)
    for (Vertex* n : v->neighbors) {
      uint64_t projected = pathDist + length(v,n);
      if (projected < n->distance) {
        n->distance = projected;
        swarm::enqueue(ssspTask,
                projected /*Timestamp*/,
                cacheLine(n) /*Hint*/, n);
      }
    }
}
```

Listing 2. Hint-tagged `sssp` task.

Each task's hint is the cache-line address of the vertex it visits. Every task iterates over its vertex's neighbor list. This incurs two levels of indirection: one from the vertex to walk its neighbor list, and another from each neighbor to access and modify the neighbor's distance. Using the line address of the vertex lets us perform all the accesses to each neighbor list from a single tile, improving locality; however, each distance is accessed from different tasks, so hints do not help with those accesses. We use cache-line addresses because several vertices reside on the same line, allowing us to exploit spatial locality.

`bfs`, `astar`, and `color` have similar structure, so we also use the visited vertex's line address as the hint. The limiting aspect of this strategy is that it fails to localize a large fraction of accesses (e.g., to `distance` in `sssp`), because each task accesses state from multiple vertices. This coarse-grain structure is natural for software implementations (e.g., sequential and parallel Galois `sssp` are written this way), but we will later see that fine-grain versions make hints much more effective.

Object IDs: In `des` and `nocsim` each task operates on one system component: a logic gate (Listing 1), or an NoC router component (e.g. its VC allocator), respectively. Similar to the graph algorithms, a task creates children tasks for its neighbors. In contrast to graph algorithms, each task only accesses state from its own component.

We tag simulator tasks with the gate ID and router ID, respectively. In `des`, using the gate ID is equivalent to using its line address, as each gate spans one line. Since each `nocsim` task operates on a router component, using component IDs or addresses as hints might seem appealing. However, components within the same router create tasks (events) for each other very often, and share state (e.g., pipeline registers) frequently. We find it is important to keep this communication local to a tile, which we achieve by using the coarser router IDs as hints.

Abstract unique IDs: In `silo`, each database transaction consists of tens of tasks. Each task reads or updates a tuple in a specific table. This tuple's address is not known at task creation time: the task must first traverse a tree to find it. Thus, unlike in prior benchmarks, hints cannot be concrete addresses. However, we know enough information to uniquely identify the tuple at task creation time: its table and primary key. Therefore, we compute the task's hint by concatenating these values. This way, tasks that access same tuple map to the same tile.

TABLE II
CONFIGURATION OF THE 256-CORE SYSTEM.

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, 4-entry ld/st 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
L3 cache	64 MB, shared, static NUCA [39] (1 MB bank/tile), 16-way, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	16×16 mesh, 128-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [72])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue entries/core (16384 total), 16 commit queue entries/core (4096 total)
Swarm instrs	5 cycles per `enqueue/dequeue/finish_task`
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [12] 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 Coalescers spill up to 15 tasks each

NOHINT and SAMEHINT: In `genome`, we do not know the data that one of its transactions, `T`, will access when the transaction is created. However, `T` spawns other transactions that access the same data as `T`. Therefore, we enqueue `T` with `NOHINT`, and its children with `SAMEHINT` to exploit parent-child locality.

Multiple patterns: Several benchmarks have different tasks that require different strategies. For instance, `kmeans` has two types of tasks: `findCluster` operates on a single point, determining its closest cluster centroid and updating the point's membership; and `updateCluster` updates the coordinates of the new centroid. `findCluster` uses the point's cache line as a hint, while `updateCluster` uses the centroid's ID. `genome` also uses a variety of patterns, as shown in Table I.

In summary, a task can be tagged with a spatial hint when some of the data it accesses can be identified (directly or abstractly) at task creation time. In all applications, integer hints are either addresses or IDs. Often, we can use either; we use whichever is easier to compute (e.g., if we already have a pointer to the object, we use addresses; if have its ID and would e.g., need to index into an array to find its address, we use IDs). It may be helpful to assign a *coarse* hint, i.e., one that covers more data than is accessed by the specific task, either to exploit spatial locality when tasks share the same cache line (e.g. `sssp`, `kmeans`), or to group tasks with frequent communication (e.g. `nocsim`).

IV. EVALUATION OF SPATIAL HINTS

A. Experimental Methodology

Modeled system: We use a cycle-accurate, event-driven simulator based on Pin [46, 53] to model Swarm systems of up to 256 cores, as shown in Fig. 1, with parameters in Table II. We use detailed core, cache, network, and main memory models, and simulate all Swarm execution overheads (e.g., running mispeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). Our configuration is similar to the 256-core Kalray MPPA [18], though with a faster clock (the MPPA is a low-power part) and about 2× on-chip memory (the MPPA uses a relatively old 28 nm process).

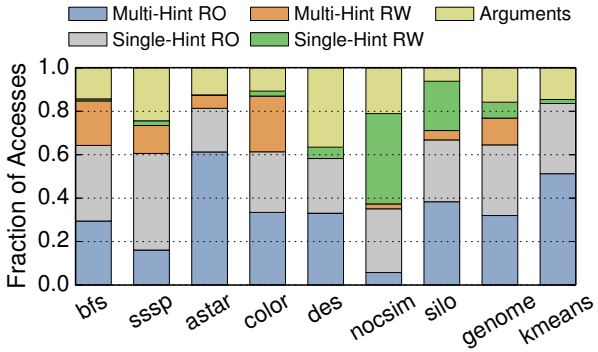


Fig. 3. Classification of memory accesses.

We model in-order, single-issue cores. Cores run the x86-64 ISA. We use the decoder and functional-unit latencies of zsim's core model, which have been validated against Nehalem [62]. Cores are scoreboarded and stall-on-use, permitting multiple memory requests in flight.

Benchmark configuration: Table I reports the input sets used. We compile benchmarks with `gcc 6.1`. All benchmarks have 1-core run-times of over 1.6 billion cycles (Table I). Benchmarks from the original Swarm paper use the same inputs. `color` operates on a YouTube social graph [45]. `nocsim` simulates a 16x16 mesh with tornado traffic at a per-tile injection rate of 0.06. STAMP benchmarks use inputs between the recommended "+" and "++" sizes, to achieve a runtime large enough to evaluate 256-core systems, yet small enough to be simulated in reasonable time. We fix the number of `kmeans` iterations to 40 for consistency across runs.

For each benchmark, we fast-forward to the start of the parallel region (skipping initialization), and report results for the full parallel region. We perform enough runs to achieve 95% confidence intervals $\leq 1\%$.

B. Effectiveness of Hints

We first perform an architecture-independent analysis to evaluate the effectiveness of hints. We profile all the memory accesses made by committing tasks, and use this to classify each memory location in two dimensions: *read-only* vs. *read-write*, and *single-hint* vs. *multi-hint*. We classify data as read-only if, during its lifetime (from allocation to deallocation time), it is read at least 1000 times per write (this includes data that is initialized once, then read widely); we classify data as single-hint if more than 90% of accesses come from tasks of a single hint. We select fixed thresholds for simplicity, but results are mostly insensitive to their specific values.

Fig. 3 classifies data accesses according to these categories. Each bar shows the breakdown of accesses for one application. We classify accesses in five types: those made to arguments,[1]

[1]Swarm passes up to three 64-bit arguments per task through registers, and additional arguments through memory; this analysis considers both types of arguments equally.

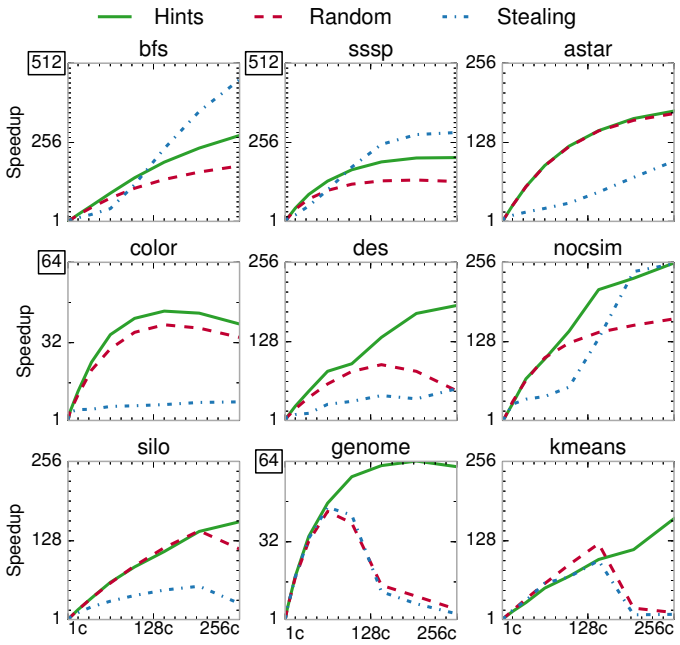


Fig. 4. Speedup of different schedulers from 1 to 256 cores, relative to a 1-core system. We simulate systems with $K \times K$ tiles for $K \leq 8$.

and those made to non-argument data of each of the four possible types (multi-/single-hint, read-only/read-write).

Fig. 3 reveals two interesting trends. First, on all applications, a significant fraction of read-only data is single-hint. Therefore, we expect hints to improve cache reuse by mapping tasks that use the same data to the same tile. All applications except `nocsim` also have a significant amount of multi-hint read-only accesses; often, these are accesses to a small amount of global data, which caches well. Second, hint effectiveness is more mixed for read-write data: in `des`, `nocsim`, `silo`, and `kmeans`, most read-write data is single-hint, while multi-hint read-write data dominates in `bfs`, `sssp`, `astar`, `color`, and `genome`. Read-write data is more critical, as mapping tasks that write the same data to the same tile not only improves locality, but also reduces conflicts.

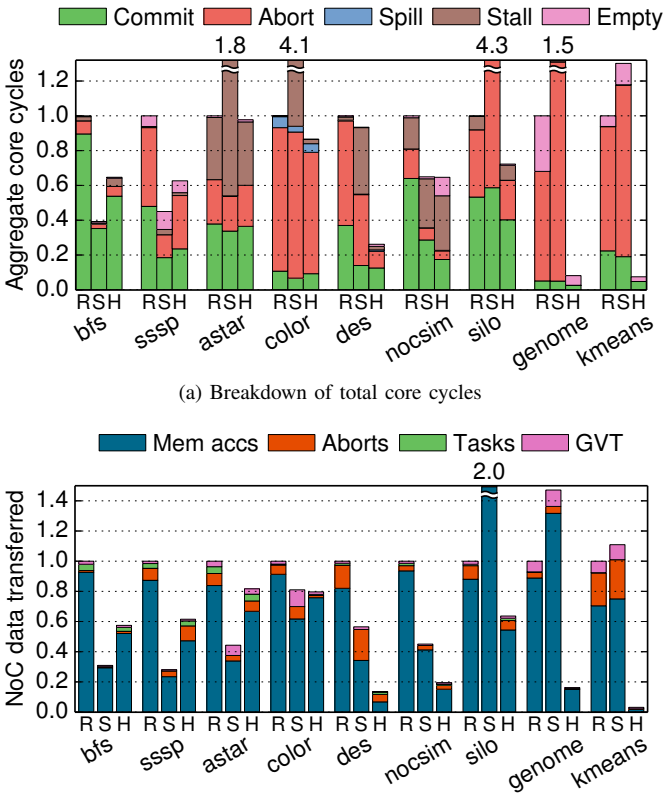
In summary, hints effectively localize a significant fraction of accesses to read-only data, and, in 4 out of 9 applications, most accesses to read-write data (fine-grain versions in Sec. V will improve this to 8 out of 9). We now evaluate the impact of these results on performance.

C. Comparison of Schedulers

Fig. 4 compares the scalability of the Random, Stealing, and Hints schedulers on 1–256-core systems, similar to Fig. 2a. As we scale the number of cores, we keep *per-core* L2/L3 sizes and queue capacities constant. This captures performance per unit area. Note that *larger systems have higher queue and cache capacities*, which sometimes causes superlinear speedups.[2]

Overall, at 256 cores, Hints performs at least as well as Random (`astar`) or outperforms it by 16% (`color`) to 13× (`kmeans`). At 256 cores, Hints scales from 39.4× (`color`) to 279× (`bfs`). Hints outperforms Random across all core counts except on `kmeans` at 16–160 cores, where Hints is hampered by imbalance (hint-based load balancing will address this). While Hints and Random follow similar trends, Stealing's performance is spotty. On the one hand, Stealing is the best scheduler in `bfs` and `sssp`, outperforming Hints by up to 65%. On the other hand, Stealing is the worst scheduler in most other ordered benchmarks, and tracks Random on unordered ones.

[2]In our prior work [37, §6.3], we analyzed the contributions of scaling queue/cache capacities in detail; here we observe the same trends.



(a) Breakdown of total core cycles

(b) Breakdown of total NoC data transferred

Fig. 5. Breakdown of (a) core cycles and (b) NoC data transferred at 256 cores, under Random, Stealing, and Hints schedulers. Each bar is normalized to Random's.

Fig. 5 gives more insight into these results by showing core cycle and network traffic breakdowns at 256 cores. Each bar of Fig. 5a shows the breakdown of cycles spent *(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 bar of Fig. 5b shows the breakdown of data transferred over the NoC (in total flits injected), including *(i)* memory accesses (between L2s and LLC, or LLC and main memory), *(ii)* abort traffic (including child abort messages and rollback memory accesses), *(iii)* tasks enqueued to remote tiles, and *(iv)* GVT updates (for commits). In both cases, results are relative to Random's. We first compare Hints and Random, then discuss Stealing.

Hints vs. Random: Beyond outperforming Random, Hints also improves efficiency, substantially reducing cycles wasted to aborted tasks and network traffic. Performance and efficiency

gains are highly dependent on the fraction of accesses to single-hint data (Sec. IV-B).

In graph analytics benchmarks, Hints achieves negligible (astar) to moderate improvements (bfs, sssp, color). bfs, sssp, and color have a substantial number of single-hint read-only accesses. These accesses cache well, reducing memory stalls. This results in a lower number of committed-task cycles and lower network traffic. However, cycles wasted on aborted tasks barely change, because nearly all accesses to contentious read-write data are multi-hint. In short, improvements in these benchmarks stem from locality of single-hint read-only accesses; since these are infrequent in astar, Hints barely improves its performance.

In des, nocsim, and silo, Hints significantly outperforms Random, from 1.4× (silo) to 3.8× (des). In these benchmarks, many read-only *and* most read-write accesses are to single-hint data. As in graph analytics benchmarks, Hints reduces committed cycles and network traffic. Moreover, aborted cycles and network traffic drop dramatically, by up to 6× and 7× (des), respectively. With Hints, these benchmarks are moderately limited by load imbalance, which manifests as stalls in nocsim and aborts caused by running too far-ahead tasks in des and silo.

Hints has the largest impact on the two unordered benchmarks, genome and kmeans. It outperforms Random by up to 13× and reduces network traffic by up to 32× (kmeans). For kmeans, these gains arise because Hints localizes and serializes all single-hint read-write accesses to the small amount of highly-contended data (the K cluster centroids). However, co-locating the many accessor tasks of one centroid to one tile causes imbalance. This manifests in two ways: *(i)* Random outperforms Hints from 16–160 cores in Fig. 4, and *(ii)* empty stalls are the remaining overhead at 256 cores. Hint-based load balancing addresses this problem (Sec. VI). In contrast to kmeans, genome has both single- and multi-hint read-write data, but Hints virtually eliminates aborts. Accesses to multi-hint read-write data rarely contend, while accesses to single-hint read-write data are far more contentious. Beyond 64 cores, both schedulers approach the limit of concurrency, dominated by an application phase with low parallelism; this phase manifests as empty cycles in Fig. 5a.

Stealing: Stealing shows disparate performance across benchmarks, despite careful tuning and idealizations (Sec. II-C). Stealing suffers from two main pathologies. First, Stealing often fails to keep tiles running tasks of roughly similar timestamps, which hurts several ordered benchmarks. Second, when few tasks are available, Stealing moves tasks across tiles too aggressively, which hurts the unordered benchmarks.

Interestingly, although they are ordered, bfs and sssp perform best under Stealing. Because most visited vertices expand the fringe of vertices to visit, Stealing manages to keep tiles balanced with relatively few steals, and keeps most tasks for neighboring nodes in the same tile. Because each task accesses a vertex and its neighbors (Listing 2), Stealing enjoys good locality, achieving the lowest committed cycles and network traffic. bfs and sssp tolerate Stealing's looser cross-tile order well, so Stealing outperforms the other schedulers.

Stealing performs poorly in other ordered benchmarks. This happens because stealing the earliest task from the most loaded tile is insufficient to keep all tiles running tasks with close-by timestamps. Instead, some tiles run tasks that are too far ahead in program order. In astar and color, this causes a large increase in commit queue stalls, which dominate execution. In des and silo, this causes both commit queue stalls and aborts, as tasks that run too early mispeculate frequently. nocsim also suffers from commit queue stalls and aborts, but to a smaller degree, so Stealing outperforms Random but underperforms Hints at 256 cores.

By contrast, genome and kmeans are unordered, so they do not suffer from Stealing's loose cross-tile order. Stealing tracks Random's performance up to 64 cores. However, these applications have few tasks per tile at large core counts, and Stealing underperforms Random because it rebalances tasks too aggressively. In particular, it sometimes steals tasks that have already run, but have aborted. Rerunning these aborted tasks at the same tile, as Random does, incurs fewer misses, as the tasks have already built up locality at the tile.

V. Improving Locality and Parallelism with Fine-Grain Tasks

We now analyze the relationship between task granularity and hint effectiveness. We show that programs can often be restructured to use finer-grain tasks, which make hints more effective.

For example, consider the coarse-grained implementation of sssp in Listing 2. This implementation causes vertex distances to be read and written from multiple tasks, which renders hints ineffective for read-write data. Instead, Listing 3 shows an equivalent version of sssp where each task operates on the data (distance and neighbor list) of a single node.

```
void ssspTaskFG(Timestamp pathDist, Vertex* v) {
 if (v->distance == UNSET) {
  v->distance = pathDist;
  for (Vertex* n : v->neighbors)
   swarm::enqueue(ssspTaskFG,
           pathDist + length(v,n) /*Timestamp*/,
           cacheLine(n) /*Hint*/, n);
 }
}
```

Listing 3. Fine-grain sssp implementation.

Instead of setting the distances of all its neighbors, this task launches one child task per neighbor. Each task accesses its own distance. This transformation generates substantially more tasks, as each vertex is often visited multiple times. In a serial or parallel version with software scheduling, the coarse-grain approach is more efficient, as a memory access is cheaper than creating additional tasks. But *in large multicores with hardware scheduling, this tradeoff reverses*: sending a task across the chip is cheaper than incurring global conflicts and serialization.

We have also adapted three other benchmarks with significant multi-hint read-write accesses. bfs and astar follow a similar approach to sssp. In color, each task operates on a vertex,

Fig. 6. Classification of memory accesses for coarse-grain (CG) and fine-grain (FG) versions.

reads from all neighboring vertices, and updates its own; our fine-grain version splits this operation into four types of tasks, each of which reads or writes at most one vertex.

We do not consider finer-grain versions of `des`, `nocsim`, `silo`, or `kmeans` because they already have negligible multi-hint read-write accesses, and it is not clear how to make their tasks smaller. We believe a finer-grain `genome` would be beneficial, but this would require turning it into an ordered program to be able to break transactions into smaller tasks while retaining atomicity [37].

Tradeoffs: In general, fine-grain tasks yield two benefits: *(i)* improved parallelism, and, *(ii)* with hints, improved locality and reduced conflicts. However, fine-grain tasks also introduce two sources of overhead: *(i)* additional work (e.g., when a coarse-grain task is broken into multiple tasks, several fine-grain tasks may need to read or compute the same data), and *(ii)* more pressure on the scheduler.

Effectiveness of Hints: Fig. 6 compares the memory accesses of coarse-grain (CG) and fine-grain (FG) versions. Fig. 6 is similar to Fig. 3, but bars are normalized to the CG version, so the height of each FG bar denotes how many more accesses it makes. Fig. 6 shows that FG versions make hints much more effective: virtually all accesses to read-write data become single-hint, and more read-only accesses become single-hint. Nevertheless, this comes at the expense of extra accesses (and work): from 8% more accesses in `sssp`, to 4.6× more in `color` (2.6× discounting arguments).

A. Evaluation

Fig. 7 compares the scalability of CG and FG versions under the three schedulers. Speedups are relative to the CG versions at one core. Fig. 8 shows cycle and network traffic breakdowns, with results normalized to CG under Random (as in Fig. 5). Overall, FG versions improve Hints uniformly, while they have mixed results with Random and Stealing.

In `bfs` and `sssp`, FG versions improve scalability and reduce data movement, compensating for their moderate increase in work. Fig. 8a shows that Hints improve locality (fewer committed cycles) and reduce aborts. As a result, FG versions under Hints incur much lower traffic (Fig. 8b), up to 4.8× lower than CG under Hints and 7.7× lower than CG under Random in `sssp`.

`astar`'s FG version does not outperform CG: though it reduces aborts, the smaller tasks stress commit queues more,

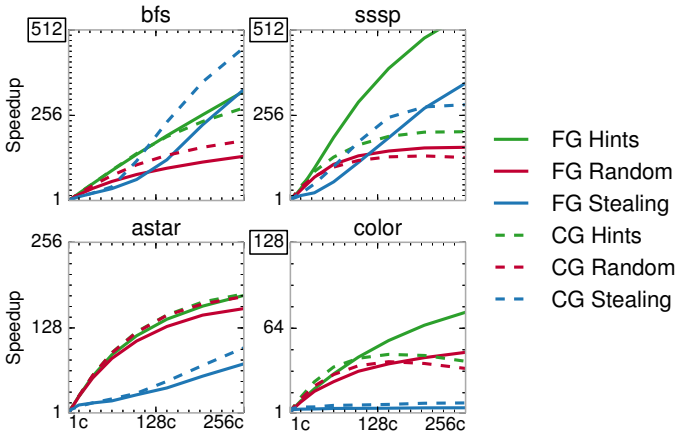
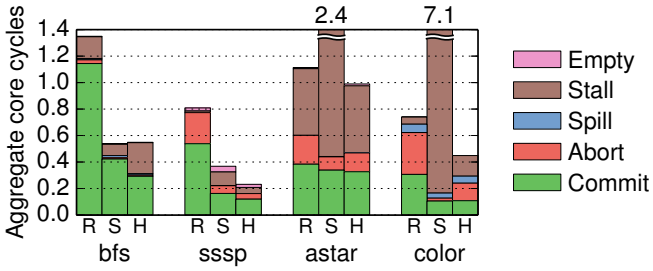
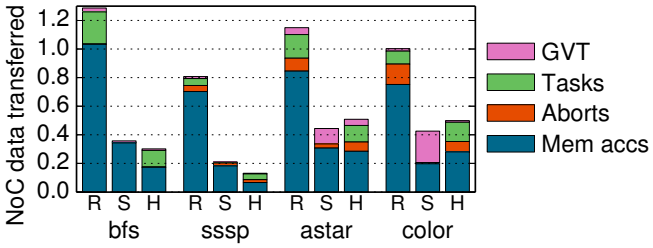


Fig. 7. Speedup of fine-grain (FG) and coarse-grain (CG) versions, relative to CG versions on a 1-core system.



(a) Breakdown of total core cycles



(b) Breakdown of total NoC data transferred

Fig. 8. Breakdown of (a) core cycles and (b) NoC data transferred in fine-grain versions at 256 cores, under Random, Stealing, and Hints. Each bar is normalized to the coarse-grain version under Random (as in Fig. 5).

increasing stalls (Fig. 8a). Nonetheless, FG improves efficiency and reduces traffic by 61% over CG under Hints (Fig. 8b).

`color`'s FG version performs significantly more work than CG, which is faster below 64 cores. Beyond 64 cores, however, FG reduces aborts dramatically (Fig. 8a), outperforming CG under Hints by 93%.

Finally, comparing the relative contributions of tasks sent in Fig. 8b vs. Fig. 5b shows that, although the amount of task data transferred across tiles becomes significant, the reduction of memory access traffic more than offsets this scheduling overhead.

In summary, fine-grain versions substantially improve the performance and efficiency of Hints. This is not always the case for Random or Stealing, as they do not exploit the locality benefits of fine-grain tasks.

VI. Data-Centric Load-Balancing

While hint-based task mapping improves locality and reduces conflicts, it may cause load imbalance. For example, in `nocsim`, routers in the middle of the simulated mesh handle more traffic than edge routers, so more tasks operate on them, and their tiles become overloaded. We address this problem by dynamically remapping hints across tiles to equalize their load.

We have designed a new load balancer because non-speculative ones work poorly. For example, applying stealing to hint-based task mapping hurts performance. The key reason is that *load is hard to measure*: non-speculative schemes use queued tasks as a proxy for load, but with speculation, underloaded tiles often do not run out of tasks—rather, they run too far ahead and suffer more frequent aborts or full-queue stalls.

Instead, we have found that the number of committed cycles is a better proxy for load. Therefore, our load balancer remaps hints across tiles to *balance their committed cycles per unit time*. Our design has three components:

1. Configurable hint-to-tile mapping with buckets: Instead of hashing a hint to produce a tile ID directly, we introduce a reconfigurable level of indirection. As shown in Fig. 9(a), when a new task is created, the task unit hashes its hint to produce a *bucket*, which it uses to index into a *tile map* and obtain the destination tile's ID, to which it sends the task.

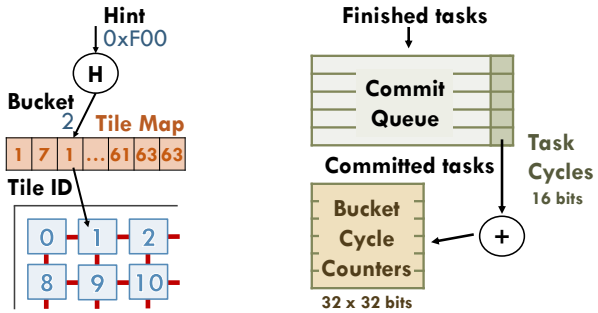
The tile map is a table that stores one tile ID for every bucket. To achieve fine-enough granularity, the number of buckets should be larger than the number of tiles. We find 16 buckets/tile works well, so at 256 cores (64 tiles) we use a 1024-bucket tile map. Each tile needs a fixed 10-bit hint-to-bucket hash function and a 1024×6-bit tile map (768 bytes).

We periodically reconfigure the tile map to balance load. The mapping is static between reconfigurations, allowing tasks to build locality at a particular tile.

2. Profiling committed cycles per bucket: Accurate reconfigurations require profiling the distribution of committed cycles across buckets. Each tile profiles cycles locally, using three modifications shown in Fig. 9(b). First, like the hashed hint (Sec. III-B), tasks carry their bucket value throughout their lifetime. Second, when a task finishes execution, the task unit records the number of cycles it took to run. Third, if the task commits, the tile adds its cycles to the appropriate entry of the per-bucket committed cycle counters.

A naive design would require each tile to have as many committed cycle counters as buckets (e.g., 1024 at 256 cores). However, each tile only executes tasks from the buckets that map to it; this number of mapped buckets is 16 per tile on average. We implement the committed cycle counters as a tagged structure with enough counters to sample 2× this average (i.e., 32 counters in our implementation). Overall, profiling hardware takes ∼600 bytes per tile.

3. Reconfigurations: Initially, the tile map divides buckets uniformly among tiles. Periodically (every 500 Kcycles in our implementation), a core reads the per-bucket committed cycle counters from all tiles and uses them to update the tile map, which it sends to all tiles.



(a) Configurable tile map (b) Profiling committed cycles

Fig. 9. Hardware modifications of hint-based load balancer.

The reconfiguration algorithm is straightforward. It computes the total committed cycles per tile, and sorts tiles from least to most loaded. It then greedily donates buckets from overloaded to underloaded tiles. To avoid oscillations, the load balancer does not seek to completely equalize load at once. Rather, an underloaded tile can only reduce its deficit (difference from the average load) by a fraction f (80% in our implementation). Similarly, an overloaded tile can only reduce its surplus by a fraction f. Reconfigurations are infrequent, and the software handler completes them in ∼50 Kcycles (0.04% of core cycles at 256 cores).

A. Evaluation

Fig. 10 reports the scalability of applications with our hint-based load balancer, denoted LBHints. LBHints improves performance on four applications, and neither helps nor hurts performance on the other five.

In `des`, LBHints outperforms Hints by 27%, scaling to 236×. As described in Sec. II-C, in `des` load imbalance causes aborts as some tiles run too far ahead; LBHints's gains come from

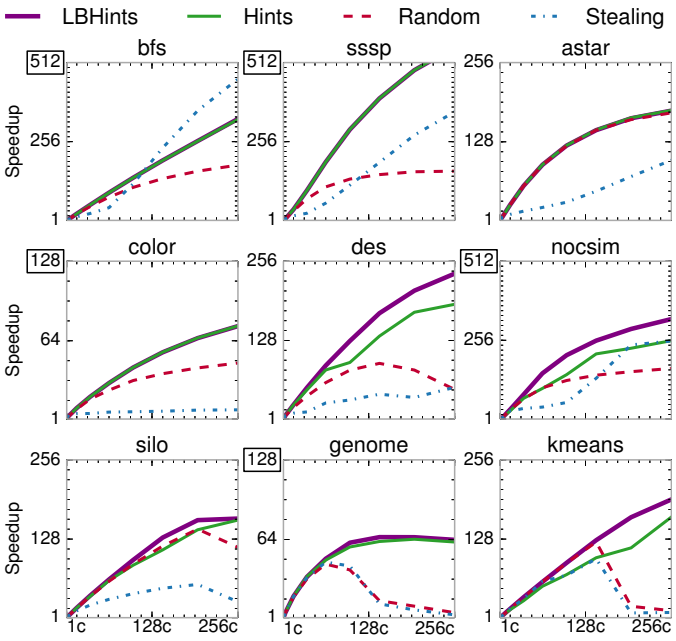


Fig. 10. Speedup of Random, Stealing, Hints, and LBHints schedulers from 1 to 256 cores, relative to a 1-core system. For applications with coarse- and fine-grain versions, we report the best-performing version for each scheme.

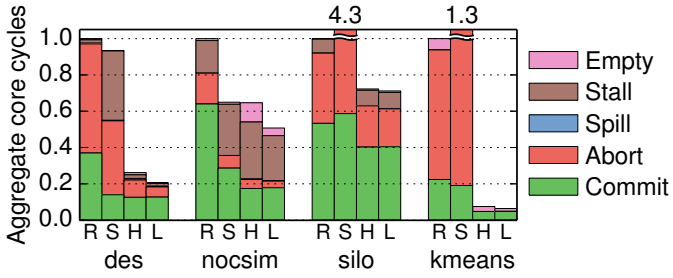


Fig. 11. Breakdown of total core cycles at 256 cores under Random, Stealing, Hints, and LBHints.

reducing aborts, as shown in Fig. 11. In `nocsim`, LBHints outperforms Hints by 27%, scaling to 325×, and in `kmeans`, LBHints outperforms Hints by 17%, scaling to 192×. These gains come from reducing empty stalls, commit queue stalls, and aborts. Finally, LBHints improves `silo` by a modest 1.5% at 256 cores, and by 18% at 144 cores. In all cases, LBHints does not sacrifice locality (same committed cycles as Hints) and yields smooth speedups.

Finally, we also evaluated using other signals as a proxy for load in the hint-based load balancer. Using the number of idle tasks in each tile to estimate load performs significantly worse than LBHints. At 256 cores, this variant improves performance over Hints by 12% on `nocsim` and 2% on `silo`, and degrades performance by 9% on `des` and 1.2% on `kmeans`. This happens because balancing the number of idle tasks does not always balance the amount of useful work across tiles.

B. Putting It All Together

Together, the techniques we have presented make speculative parallelism practical at large core counts. Across our nine applications, Random achieves 58× gmean speedup at 256 cores; Hints achieves 146×; with the fine-grain versions from Sec. V instead of their coarse-grain counterparts, Hints scales to 179× (while Random scales to 62× only); and LBHints scales to 193× gmean.[3]

VII. Related Work

A. Scheduling in Speculative Parallelism

Speculative execution models have seen relatively little attention with regards to optimizing locality.

Ordered parallelism: Thread-Level Speculation (TLS) schemes dispatch tasks to threads as they become available, without concern for locality [24, 27, 57, 58, 66, 68]. TLS schemes targeted systems with few cores, but cache misses hinder TLS scalability even at small scales [23].

Unordered parallelism: TM programs are commonly structured as threads that execute a fixed sequence of transactions [13, 28, 49]. Prior work has observed that it is often beneficial to structure code as a collection of transactional tasks, and schedule them across threads using a variety of hardware and software techniques [5, 6, 8, 9, 19, 21, 61, 77]. Prior transactional schedulers focus on *limiting concurrency*, not spatial task mapping. These schemes are either reactive or predictive. ATS [77], CAR-STM [20], and Steal-on-Abort [5] serialize aborted transactions after the transaction they conflicted with, avoiding repeated conflicts. PTS [8], BFGTS [9], Shrink [21], and HARP [6] instead predict conflicts by observing the read- and write-sets of prior transactions, and serialize transactions that are predicted to conflict. Unlike predictive schemes, we avoid conflicts by leveraging program hints. Hints reduce conflicts more effectively than prior predictive schedulers, and require much simpler hardware. Moreover, unlike prior transactional schedulers, our approach does not rely on centralized scheduling structures or frequent broadcasts, so it scales to hundreds of cores.

A limitation of our approach vs. predictive transaction schedulers is that programmers must specify hints. We have shown that it is easy to provide accurate hints. It may be possible to automate this process, e.g. through static analysis or profile-guided optimization; we defer this to future work.

Data partitioning: Kulkarni et al. [41] propose a software speculative runtime that exploits partitioning to improve locality. Data structures are statically divided into a few coarse partitions, and partitions are assigned to cores. The runtime maps tasks that operate on a particular partition to its assigned core, and reduces overheads by synchronizing at partition granularity. Schism [16] applies a similar approach to transactional databases. These techniques work well only when data structures can be easily divided into partitions that are both balanced and capture most parent-child task relations, so that most enqueues do not leave the partition. Many algorithms do not meet these conditions. While we show that simple hint assignments that do not rely on careful static partitioning work well, more sophisticated mappings may help some applications. For example, in `des`, mapping adjacent gates to nearby tiles may reduce communication, at the expense of complicating load balancing. We leave this exploration to future work.

Distributed transactional memory: Prior work has proposed STMs for distributed systems [11, 32, 60]. Some of these schemes, like ClusterSTM [11], allow migrating a transaction across machines instead of fetching remotely-accessed data. However, their interface is more onerous than hints: in ClusterSTM, programmers must know exactly how data is laid out across machines, and must manually migrate transactions across specific processors. Moreover, these techniques are dominated by the high cost of remote accesses and migrations [11], so load balancing is not an issue.

B. Scheduling in Non-Speculative Parallelism

In contrast to speculative models, prior work for non-speculative parallelism has developed many techniques to improve locality, often tailored to specific program traits [2, 4, 10, 15, 33, 50, 65, 74, 76]. Work-stealing [2, 10] is the most widely used technique. Work-stealing attempts to keep parent and child tasks together, which is near-optimal for divide-and-conquer algorithms, and as we have seen, minimizes data movement in some benchmarks (e.g., `bfs` and `sssp` in Sec. IV).

[3]The corresponding harmonic-mean speedups are 25× for Random, 117× for Hints, 146× for Hints with fine-grained versions, and 154× for LBHints.

Due to its low overheads, work-stealing is the foundation of most parallel runtimes [22, 34, 40], which extend it to improve locality by stealing from nearby cores or limiting the footprint of tasks [2, 26, 63, 76], or to implement priorities [44, 51, 63]. Prior work within the Galois project [31, 44, 54] has found that irregular programs (including software-parallel versions of several of our benchmarks) are highly sensitive to scheduling overheads and policies, and has proposed techniques to synthesize adequate schedulers [51, 55]. Likewise, we find that work-stealing is sensitive to the specific policies it uses.

In contrast to these schemes, we have shown that a simple hardware task scheduling policy can provide robust, high performance across a wide range of benchmarks. Hints enable high-quality spatial mappings and produce a balanced work distribution. Hardware task scheduling makes hints practical. Whereas a software scheduler would spend hundreds of cycles per remote enqueue on memory stalls and synchronization, a hardware scheduler can send short tasks asynchronously, incurring very low overheads on tasks as small as few tens of instructions. Prior work has also investigated hardware-accelerated scheduling, but has done so in the context of work-stealing [42, 64] and domain-specific schedulers [25, 73].

VIII. Conclusion

We have presented spatial hints, a general technique that leverages application-level knowledge to achieve high-quality spatial task mappings in speculative programs. A hint is an abstract value, given at task creation time, that denotes the data likely to be accessed by a task. We have enhanced Swarm, a state-of-the-art speculative architecture, to exploit hints by *(i)* running tasks likely to access the same data on the same tile, *(ii)* serializing tasks likely to access the same data, and *(iii)* balancing work across tiles in a locality-aware fashion. We have also studied the relationship between task granularity and locality, and shown that programs can often be restructured to use finer-grain tasks to make hints more effective.

Together, these techniques make speculative parallelism practical on large-scale systems: at 256 cores, the baseline Swarm system accelerates nine challenging applications by 5–180× (gmean 58×). With our techniques, speedups increase to 64–561× (gmean 193×). Beyond improving gmean performance by 3.3×, our techniques make speculation more efficient, reducing aborted cycles by 6.4× and network traffic by 3.5× on average.

Acknowledgments

We sincerely thank Nathan Beckmann, Harshad Kasture, Po-An Tsai, Guowei Zhang, Anurag Mukkara, Li-Shiuan Peh, Virginia Chiu, Yee Ling Gan, and the anonymous reviewers for their helpful feedback. William Hasenplaugh and Chia-Hsin Chen graciously shared the serial code for the `color` [30] and `nocsim` benchmarks. 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 Jeffrey was partially supported by an NSERC postgraduate scholarship.

References

[1] "9th DIMACS Implementation Challenge: Shortest Paths," 2006.
[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *SPAA*, 2000.
[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *ISPASS*, 2009.
[4] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of X10 computations with bounded resources," in *SPAA*, 2007.
[5] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *HiPEAC*, 2009.
[6] A. Armejach, A. Negi, A. Cristal, O. Unsal, P. Stenstrom, and T. Harris, "HARP: Adaptive abort recurrence prediction for hardware transactional memory," in *HiPC-20*, 2013.
[7] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *10th DIMACS Implementation Challenge Workshop*, 2012.
[8] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *MICRO-42*, 2009.
[9] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom filter guided transaction scheduling," in *MICRO-44*, 2011.
[10] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, 1999.
[11] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *PPoPP*, 2008.
[12] J. Carter and M. Wegman, "Universal classes of hash functions (extended abstract)," in *STOC-9*, 1977.
[13] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *ISCA-33*, 2006.
[14] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *HPCA-13*, 2007.
[15] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas *et al.*, "Scheduling threads for constructive cache sharing on CMPs," in *SPAA*, 2007.
[16] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *VLDB*, 2010.
[17] T. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM TOMS*, vol. 38, no. 1, 2011.
[18] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *HPEC*, 2013.
[19] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic Scheduling for Hardware Transactional Memory," in *SPAA*, 2015.
[20] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *PODC-27*, 2008.
[21] A. Dragojević, R. Guerraoui, A. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *PODC-28*, 2009.
[22] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *IWOMP-4*, 2008.
[23] S. L. Fung and J. G. Steffan, "Improving cache locality for thread-level speculation," in *IPDPS*, 2006.
[24] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas, "Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors," in *HPCA-9*, 2003.
[25] J. P. Grossman, J. S. Kuskin, J. A. Bank, M. Theobald, R. O. Dror, D. J. Ierardi, R. H. Larson, U. B. Schafer, B. Towles *et al.*, "Hardware support for fine-grained event-driven computation in Anton 2," in *ASPLOS-XVIII*, 2013.
[26] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *IPDPS*, 2010.
[27] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS-VIII*, 1998.
[28] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *ISCA-31*, 2004.
[29] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybernetics*, vol. 4, no. 2, 1968.
[30] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *SPAA*, 2014.

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

[32] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Distributed Computing*, 2005.

[33] B. Holt, P. Briggs, L. Ceze, and M. Oskin, "Alembic: automatic locality extraction via migration," in *OOPSLA*, 2014.

[34] Intel TBB, "http://www.threadingbuildingblocks.org."

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

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

[37] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A Scalable Architecture for Ordered Parallelism," in *MICRO-48*, 2015.

[38] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "Unlocking Ordered Parallelism with the Swarm Architecture," *IEEE Micro*, vol. 36, no. 3, 2016.

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

[40] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Scheduling strategies for optimistic parallel execution of irregular programs," in *SPAA*, 2008.

[41] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," in *ASPLOS-XIII*, 2008.

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

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

[44] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par*, 2015.

[45] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, jun 2014.

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

[47] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys*, vol. 48, no. 2, 2015.

[48] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC*, 2008.

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

[50] J. E. Nelson, "Latency-Tolerant Distributed Shared Memory For Data-Intensive Applications," Ph.D. dissertation, 2015.

[51] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *ASPLOS-XVI*, 2011.

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

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

[54] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich *et al.*, "The tao of parallelism in algorithms," in *PLDI*, 2011.

[55] D. Prountzos, R. Manevich, and K. Pingali, "Synthesizing parallel graph programs via automated planning," in *PLDI*, 2015.

[56] X. Qian, W. Ahn, and J. Torrellas, "ScalableBulk: Scalable cache coherence for atomic blocks in a lazy environment," in *MICRO-43*, 2010.

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

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

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

[60] M. M. Saad and B. Ravindran, "Hyflow: A high performance distributed software transactional memory framework," in *HPDC-20*, 2011.

[61] D. Sainz and H. Attiya, "RELSTM: A proactive transactional memory scheduler," in *TRANSACT*, 2013.

[62] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *ISCA-40*, 2013.

[63] D. Sanchez, D. Lo, R. M. Yoo, J. Sugerman, and C. Kozyrakis, "Dynamic Fine-Grain Scheduling of Pipeline Parallelism," in *PACT-20*, 2011.

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

[65] H. Simhadri, G. Blelloch, J. Fineman, P. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," in *SPAA*, 2014.

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

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

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

[69] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan, "GRAMPS: A programming model for graphics pipelines," *ACM Trans. Graph.*, vol. 28, no. 1, 2009.

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

[71] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, 1967.

[72] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, no. 5, 2007.

[73] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[74] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger *et al.*, "Productivity and performance using partitioned global address space languages," in *PASCO*, 2007.

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

[76] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: A quantitative limit study," in *SPAA*, 2013.

[77] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *SPAA*, 2008.