A Scalable Architecture for Reprioritizing Ordered Parallelism

Gilead Posluns
University of Toronto
Canada
gil.posluns@mail.utoronto.ca

Yan Zhu
University of Toronto
Canada

Guowei Zhang
Huawei
China
zhangguowei9@hisilicon.com

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

ABSTRACT

Many algorithms schedule their work, or tasks, according to a priority order for correctness or faster convergence. While priority schedulers commonly implement task *enqueue* and *dequeueMin* operations, some algorithms need a *priority update* operation that alters the scheduling metadata for a task. Prior software and hardware systems that support scheduling with priority updates compromise on either parallelism, work-efficiency, or both, leading to missed performance opportunities. Moreover, incorrectly navigating these compromises violates correctness in those algorithms that are not resilient to relaxing priority order.

We present Hive, a task-based execution model and multicore architecture that extracts abundant fine-grain parallelism from algorithms with priority updates, while retaining their strict priority schedules. Like prior hardware systems for ordered parallelism, Hive uses data- and control-dependence speculation and a large speculative window to execute tasks in parallel and out of order. Hive improves on prior work by *(i)* directly supporting updates in the interface, *(ii)* identifying the novel *scheduler-carried dependence*, and *(iii)* speculating on such dependences with *task versioning*, distinct from data versioning. Hive enables safe speculative updates to the schedule and avoids spurious conflicts among tasks to better utilize speculation tracking resources and efficiently uncover more parallelism. Across a suite of nine benchmarks, Hive improves performance at 256 cores by up to 2.8× over the next best hardware solution, and even more over software-only parallel schedulers.

CCS CONCEPTS

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

KEYWORDS

priority scheduling, priority updates, task-level parallelism, ordered irregular parallelism, speculative execution

ACM Reference Format:
Gilead Posluns, Yan Zhu, Guowei Zhang, and Mark C. Jeffrey. 2022. A Scalable Architecture for Reprioritizing Ordered Parallelism. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3470496.3527387

ISCA '22, June 18–22, 2022, New York City, NY

1 INTRODUCTION

The optimal or fast-converging algorithms for many problems require their work items, or tasks, to execute according to some priority order. Sequential implementations use a priority queue to schedule the tasks. In particular, we focus on those algorithms that dynamically alter the task schedule with *priority update* operations [28, 76] that associate an object ID with a priority. For example, some graph algorithms will assign an initial priority to every vertex (ID). Their executions consist of processing the highest-priority vertex, updating the priorities of its neighbors, and then repeating the process in a loop with the next highest priority vertex.

Ideally, these algorithms should have abundant task-level parallelism, as true data dependences among tasks (loop iterations) are rare for sparse data structures like graphs. However, practically extracting this parallelism is challenging as it requires *(i)* ensuring irregular data dependences flow in the required order and *(ii)* circumventing the false data dependences on the global scheduling structure, which every task would otherwise read and write. Software and hardware systems exploit this ordered irregular parallelism [58] using one or more of three techniques: bulk-synchronous parallelism, speculative parallelism, or relaxing the order.

Current software parallel frameworks strive to drive down scheduling overheads. Bucketing [23, 88] is a bulk-synchronous approach that executes groups of equal-priority tasks (buckets) in parallel. Bucketing can retain a strict priority order, giving work-efficient implementations, but the barriers between buckets limit parallelism when there is little work per bucket. Moreover, priority updates on the schedule can create even more buckets, further constraining parallelism. Speculative techniques [14, 33, 34, 44] uncover parallelism *across priorities*, speculating that tasks will access independent data, circumventing barriers. However, speculation overheads in software overwhelm the benefits of inter-priority parallelism for small tasks [33, 34]. Schedulers that *relax* the priority order [6, 7, 46, 56, 60, 64, 66, 80, 85, 89] present a middle ground between these techniques, providing a best effort to dispatch tasks in order, but with only probabilistic guarantees, if any. Approximating the desired task order avoids barriers and enables distributing software queues to reduce contention due to scheduling. However, relaxation is only amenable to those algorithms where task ordering is not required for correctness, but instead reduces redundant work to improve convergence time [4, 5, 51, 56]. Moreover, the higher the core count, the greater the deviation from the desired priority order, worsening work efficiency and convergence time [8, 66].

Prior order-aware hardware systems are subject to the same parallelism vs. relaxation trade-off, or do not support priority update operations. PolyGraph [21] provides relaxed priority semantics or accelerated bulk-synchronous execution, but does not provide a scalable strict priority schedule with update semantics. Thread-level speculation [30, 42, 62, 63, 70, 71, 75] targets the automatic

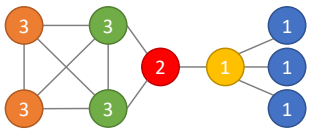


Figure 1: Graph labeled with the coreness of each vertex.

```
1 PriorityQueue pq;
2 for (int v : G.V)
3   pq.enqueue(v, G.degree[v]);
4 while (!pq.empty()) {
5   int v, int prio = pq.dequeueMin();
6   coreness[v] = prio;
7   for (int nbr : G.edges[v])
8     if (pq.getPrio(nbr) > prio)
9       pq.decrementPrio(nbr);
10 }
```

Listing 1: Sequential code for kcore.

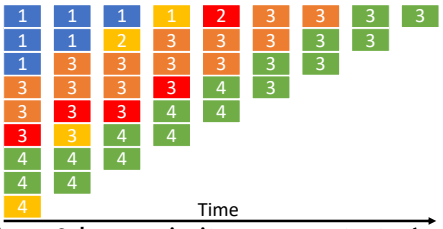


Figure 2: kcore priority queue contents. A vertex is dequeued from the top at each time step.

parallelization of sequential code, so couples loop iteration order to execution order. Consequently, to schedule new work to execute at a future time, a scheduling structure in software is still required, serializing all tasks with dependences through the scheduler. In contrast, hardware for speculative ordered parallelism, such as Swarm [37, 38], Fractal [72], and Chronos [3], can implement a dynamic strict priority schedule. However, the sequential queue these systems abstract supports only *enqueue* and *dequeueMin* operations, notably not a priority update. Instead, a programmer wishing to implement an algorithm needing priority updates must add software scheduling structures that shadow the hardware ones, tracking much of the same state redundantly, and they must write early exiting tasks to emulate the sequential behavior. Such programs clog the speculative task-tracking data structures of the hardware, resulting in stalls and reduced throughput.

This paper presents Hive, the first execution model and speculative multicore architecture to express and extract parallelism from ordered algorithms with priority updates. We characterize the implications of a strict priority schedule with updates, and the complex *scheduler-carried dependences* created between successive tasks in the schedule (Sec. 2, Sec. 4). The Hive execution model enables the programmer to convey the desired priority schedule (and updates) directly to hardware, abstracting a strict priority queue (Sec. 3). We present a taxonomy of the ways in which several algorithms use priority updates (Sec. 3.3). Our Hive implementation (Sec. 5) adds modest area to the Swarm architecture, extracting parallelism from the abstract queue by speculatively executing tasks out of priority order. Importantly, Hive introduces *task versioning*, a method of speculating on scheduler dependences, similarly to how memory versioning enables data dependence speculation (Sec. 4).

This work makes three key contributions:

- A description of the unique class of dependence between operations on a priority queue supporting priority updates.
- A new technique to enable safe speculation on the state of a priority scheduler that is supporting speculative parallelism, while being speculatively updated.
- An execution model and hardware system supporting reprioritizable ordered parallelism, which achieves up to 2.8× speedup (gmean 52%) over hardware supporting ordered parallelism without priority updates, and more over software-only approaches.

2 MOTIVATION

The optimal algorithm for the k-core decomposition problem [50, 65] (`kcore`) requires a strict priority schedule supporting priority updates, and illustrates the challenges and opportunities in this work. The maximum core, or *coreness*, of a vertex is an important property in a variety of domains, including graph mining [67], graph visualization [10], statistical mechanics [53], ecosystem analysis [54], and bioinformatics [81]. The coreness can represent the importance of a vertex [49] or its position in a hierarchical description of the graph [10]. A k-core of an undirected graph is a maximal set of vertices where every vertex has at least k edges to other vertices in the k-core. For example, the graph in Fig. 1 has a 3-core consisting of the green and orange vertices in a square. The coreness of a vertex is the highest k for which it is part of a k-core. The yellow vertex has degree 4, but it only has a coreness of 1 because 3 of its edges are to vertices that are not part of a 2-core. The red vertex has a coreness of 2, because it has 2 edges to vertices that are part of a k-core for $k \geq 2$.

Listing 1 shows the sequential algorithm for `kcore`, which determines the coreness of each vertex in the graph. In essence, for each increasing value of k, the algorithm recursively removes all vertices with degree k, then repeats with the next value of k, until no vertices remain. The k value at which a vertex is removed is its *coreness*. This implementation enqueues each vertex into a queue with priority initially equal to its degree (line 3). It then runs a loop that dequeues one vertex from the queue each iteration (line 5). Each vertex is dequeued exactly once, at which point it decrements the priority (which is tracking the remaining degree) of all of its neighbors that are still in the queue (lines 7-9). When a vertex is dequeued there exist no remaining vertices in the queue with lower degree, so its current degree (and priority) is its coreness.

This algorithm depends on three priority queue operations. In addition to the `enqueue` and `dequeueMin` operations typical of any priority queue [79] (e.g., the `C++ std::priority_queue`), it also uses `decrementPrio` [28, 76] (e.g., in the `boost fibonacci_heap`). [1] The latter operation indexes into the queue by vertex (ID) and updates its priority to dynamically alter the vertex's position in the schedule.

The priority schedule of `kcore` is required for correctness. For example, Fig. 2 shows the contents of the priority queue over time for the graph in Fig. 1. The yellow vertex is initially scheduled last, but the blue vertex iterations each decrement its priority one after the other until yellow runs immediately after them. The yellow iteration then decrements the priority of the red vertex, rather than vice versa. Vertices with equal priority, such as the blue and green/orange sets, get an arbitrary order in the queue. If the algorithm were instead to process the red vertex before the yellow, then it would incorrectly assign red a coreness of 3 instead of 2.

There is ample parallelism available in `kcore`, once the false data dependences on the scheduling structure [37] are abstracted away. In the running example, iterations that operate on the blue and

[1] This code adds a fourth `getPrio` command for readability.

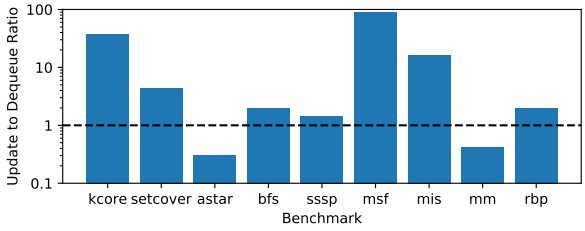


Figure 3: Ratio of scheduler updates to dequeues at 1 core.

orange vertices are independent and could therefore be processed in parallel. However, current software and hardware struggles to efficiently unlock this parallelism for the following reasons.

Priority updates often outnumber dequeues: Since vertices on average have more than one neighbor, then `kcore` will at least consider calling `decrementPrio` more than it calls `dequeueMin`. Fig. 3 shows the high ratio of conditional updates vs. dequeues on large inputs for our benchmark suite of nine algorithms, including `kcore` (see Sec. 6.1 for methodology). The ratio is greater than 1 for all benchmarks except `astar`, which terminates before exploring the entire graph, and `mm`, where priority updates can only eliminate two thirds of the fine-grain tasks [36], capping the ratio at 1. This typically high ratio is significant because for most algorithms in our suite, including `kcore`, each loop iteration is short with little additional processing beyond the priority updates (Table 4), and the final priorities of the vertices *are* the output of the algorithm. Since priority updates can comprise the majority of an algorithm's work, they must be performed in a scalable and efficient way.

Priority updates cause poor performance in software: Like `dequeueMin` and `enqueue`, priority updates are read-modify-write operations on the scheduling structure, so they contend when performed on shared global state. Software schedulers with privatization [46, 56, 66, 85] mitigate some contention, but they consequently do not update the global scheduler immediately. Therefore, to maintain `kcore`'s strict priority schedule, systems such as Julienne [23] and Ordered GraphIt [88] use bulk-synchronous parallelism among equal-priority tasks and apply reductions to the privatized queues into a consistent state at barriers. These systems can only extract parallelism from the potentially limited work between barriers, and are unable to extract parallelism across priorities.

Fig. 4 shows two views of `kcore`'s work distribution over bucket sizes. The right side of the blue CDF shows that a few barriers have massive available work (parallelism). However, the left side of the orange PDF reveals an Amdahl bottleneck: 10% of all work happens between barriers with 2000 or fewer vertices per barrier, and nearly

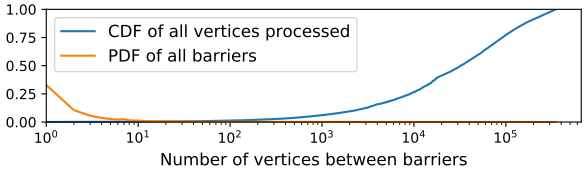


Figure 4: `kcore` work distribution for Julienne [23]. Blue shows the CDF of all vertices processed (y-axis) with increasing vertices per barrier (x-axis). Orange shows the PDF of all barriers (y-axis) with a given number of vertices (x-axis).

one third of all barriers process a single vertex. Julienne and GraphIt are entirely sequential for this large number of iterations.

To extract parallelism across priorities, the alternatives to synchronous execution are relaxation, speculation, and task dependence graphs. Relaxed priority queues are ineligible for `kcore` because relaxed scheduling can lead to incorrect outputs. `kcore`'s tiny tasks and abundant updates would lead to high overheads in scheduler-aware speculation [33, 44], and kinetic dependence graphs [34], as observed in similar algorithms.

Priority updates have poor performance in hardware: To reduce the scheduling overheads of software, hardware systems have been proposed for extracting parallelism from applications with priority-ordered tasks. However, they do not support explicit priority update operations without either relaxing the schedule, or requiring schedule tracking metadata in software.

PolyGraph [21] is a graph accelerator that offers a choice between synchronous execution—hardware acceleration of the Julienne and Ordered GraphIt approach—and asynchronous execution that removes barriers by providing a relaxed priority schedule. PolyGraph supports priority updates through task coalescing, but only when a relaxed schedule is allowed and tasks will not spill to memory.

Swarm [37] and Chronos [3] provide strict priority scheduling, using hardware speculation to extract parallelism across priorities, but lack builtin support for priority updates. A programmer wishing to write a program with priority updates must *(i)* implement a scheduling metadata structure in software to track the current priority of each object, and *(ii)* restructure task code to check this metadata and exit early if the task's object priority has been updated, making the task moot. This is similar to writing Listing 1 with a priority queue that only supports *enqueue* and *dequeueMin*, as shown in Listing 2. This code is largely similar to Listing 1, with the exceptions of lines 10 and 12-14. Line 10 checks the priority-tracking metadata structure to ensure that vertex `v` was not already processed at an earlier priority. If it was, the loop exits that task early and moves on to the next vertex. Because the queue does not support priority updates, lines 12-14 instead conditionally decrement the priority of `v`'s neighbors in the scheduling metadata and enqueue a new task for each vertex `nbr` at its new priority. When the old later-ordered task dequeues, it will exit early at line 10.

Having tasks check a condition and potentially do nothing is similar to predication of instructions as an alternative to conditional

```
1 PriorityQueue pq;
2 int prios[G.n]; // Scheduling metadata
3 for (int v : G.V) {
4   prios[v] = G.degree[v];
5   pq.enqueue(v, prios[v]);
6 }
7 while (!pq.empty()) {
8   int v, int prio = pq.dequeueMin();
9   // Skip if this iteration/task is moot
10  if (prio > prios[v]) continue;
11  for (int nbr : G.edges[v])
12    if (prios[nbr] > prio) {
13      prios[nbr]--;
14      pq.enqueue(nbr, prios[nbr]);
15    }
16 }
17 coreness = prios
```

Listing 2: Sequential `kcore` without priority updates.

branches [9]. Like instruction-level predication, when overused [59], early exiting tasks fill the speculation state-tracking structures with tasks that are practically NOPs. Since the ratio of priority updates to dequeues is so high, this is exactly what happens when implementing priority updates in these systems. Although task-level predication is a valid approach, we advocate for a more expressive execution model and hardware support to better utilize on-chip resources for extracting reprioritizable ordered parallelism.

3 HIVE EXECUTION MODEL

A Hive program consists of priority-ordered *tasks* which can be logically bound to *objects* to enable updates to the schedule. Hive hardware (Sec. 4, Sec. 5), extracts speculative parallelism across hundreds of cores by finding independent tasks to run out of order. However, Hive guarantees the program output will always match that of a sequential thread scheduling the tasks in a priority queue supporting updates [28, 76]. Every task can read and write arbitrary shared memory, can dynamically *update* tasks bound to objects, and can *enqueue* new tasks unbounded from any object. The update and enqueue operations assign each task an integer *timestamp* which encodes its priority order in the modeled queue. Hive objects are the program's core data type for scheduling, and each object is identifiable with a unique number, such as a memory address or ID. The programmer defines objects as needed in application memory. Hive records the binding for every object ID to one or no queued task in an *object table* residing in a protected region of memory, inaccessible to the tasks except through an API (Sec. 3.1). When no queued task is bound to an object, the object table instead records the timestamp of the object's last queued task (or infinity if none).

Hive ensures that tasks *appear to* execute in increasing timestamp order, as if a sequential loop repeatedly dequeues the lowest-timestamp task from the modeled queue, runs it, then dequeues the next task, until the queue is empty. Tasks with equal timestamp are atomic, being serialized arbitrarily among each other. This execution model has two key consequences: *(i)* a task's enqueued children appear to execute only after the parent finishes—children are ordered after the parent—and *(ii)* a task's accesses to shared memory and its updates to object-task binding appear to happen atomically and before the next task in priority order would be dequeued.

```
1 void removeV(Vertex* v, Timestamp ts) { // Task
2   for (Vertex* nbr : v->neighbors())
3     hive::decrTS(&removeV, nbr, 1);
4 }
5 void main(int argc, char** argv) {
6   hive::init(G.n);
7   for (Vertex* v : G.V)
8     hive::update(&removeV, v, v->degree);
9   hive::run();
10  hive::extract(coreness);
11 }
```

Listing 3: Hive implementation of `kcore`

3.1 Programming interface

Table 1 shows the Hive API, which programs use to enqueue tasks and manipulate the object-task bindings. Listing 3 shows the API in action with a Hive implementation of `kcore`.

A Hive task is an instance of a function with timestamp and arguments received through registers. Listing 3 defines one task function, `removeV`, which logically removes a vertex `v` (a Hive *object*) from the graph, performing the work of lines 6-9 of Listing 1.

Any task can set, `update`, or `cancel` tasks bound to objects, by calling the given (inlined) Hive functions with a task function pointer, timestamp, and arguments. These are passed to hardware through registers with one new instruction (Sec. 5). While these *basic* operations provide sufficient schedule manipulation for some programs, others require *timestamp-relative* task updates that depend on the timestamp of the task currently bound to an object. One could implement such relative updates with `hive::getTS` and `hive::update`, but *(i)* this complicates programming, and *(ii)* it hurts performance with remote accesses to the object table in memory. The Hive interface improves expressiveness by adding min, increment, and decrement updates, similar to those in DSLs [88].

Listing 3 uses both basic and relative task updates. The `main` function binds an initial `removeV` task to every vertex by calling `hive::update` with timestamp equal to `v`'s original degree. The `removeV` task itself decrements the task timestamp (degree) for all neighbors of its vertex `v`. Since `kcore` tasks are ordered by their current degree, `removeV` implicitly sets `v`'s coreness in the object table as the last (and only) timestamp for a task that executed on `v`.

Table 1: Hive programming interface

		Signature	Description
	`void`	`hive::init<flags>(nobjs)`	Reserve object table capacity for `nobjs` objects with no initially queued tasks. Empty (null) tasks implicitly have timestamp infinity.
	`void`	`hive::extract(Timestamp* dest)`	Extract the timestamp of the last task that executed for every object ID.
Basic	`void`	`hive::update(taskFn, oid, ts, args...)`	Replace or set the queued task bound to object `oid`.
Basic	`void`	`hive::cancel(oid)`	Remove the queued task bound to object `oid`. This is either an `update` with timestamp infinity, or an `updateMin` with timestamp 0 and an empty task.
Basic	`Timestamp`	`hive::getTS(oid)`	Return the timestamp of the task currently or last executed bound to `oid`.
Basic	`void`	`hive::enqueue(taskFn, ts, args...)`	Queue a task unbounded from any object
TS-Relative	`void`	`hive::updateMin(taskFn, oid, ts, args...)`	Replace the queued task bound to object `oid` only if `ts < hive::getTS(oid)`.
TS-Relative	`void`	`hive::incrTS(taskFn, oid, delta, args...)`	Replace the queued task bound to object `oid`, adding a signed `delta` to the previous timestamp. NOP if there is no task bound to the object.
TS-Relative	`void`	`hive::decrTS(taskFn, oid, delta, args...)`	Replace the queued task bound to object `oid`, where the new timestamp is equal to the max of the previous timestamp minus an unsigned `delta` and the caller's timestamp, only if doing so would decrease the timestamp. NOP otherwise, or if there is no task bound to the object.

A program invokes Hive by initializing the object table (`hive::init`), enqueuing or updating some initial task(s), then calling `hive::run`, which returns control to the main thread when there are no more tasks to run. Listing 3 initializes the object table with one entry for every vertex in the graph. In many algorithms, the program output is in fact the priority at which every task ran, so Hive provides the `hive::extract` function to copy the last timestamp for every object to a buffer in application memory.

3.2 Example modeled execution

Fig. 5 illustrates how Hive *appears to* serially execute the `kcore` implementation in Listing 3 on the graph in Fig. 1. Task execution is shown as a circle and a priority update to an object's task is shown as a rounded box. Every vertex (object on the y-axis) has initial timestamp equal to its degree. At each logical time step (x-axis) Hive hardware dequeues and executes the queued task with lowest current timestamp. The three blue vertices have equal initial timestamp of 1, so are dequeued in arbitrary order among themselves, but execute atomically. Importantly, *the priority update performed by each task takes effect atomically with the task, even before the next apparent dequeue.* Timestamp decrements do not commute with dequeue, just as counter decrements do not commute with a read [87]. The three blue tasks decrement the yellow vertex task timestamp down to 1. Despite initializing with a large timestamp, the yellow task is next to execute due to the priority updates of this algorithm. The yellow task decrements the red vertex priority to 2, but notably does *not* set a new task for its blue vertex neighbors (dashed rounded box). This is because its neighbors have already executed with equal or lower timestamp, so the semantics of `decrTS` state that no new task is created in this case. The red vertex then dequeues with the lowest timestamp, updates its green neighbor timestamps, but does not update the yellow neighbor. The execution proceeds until no queued tasks remain.

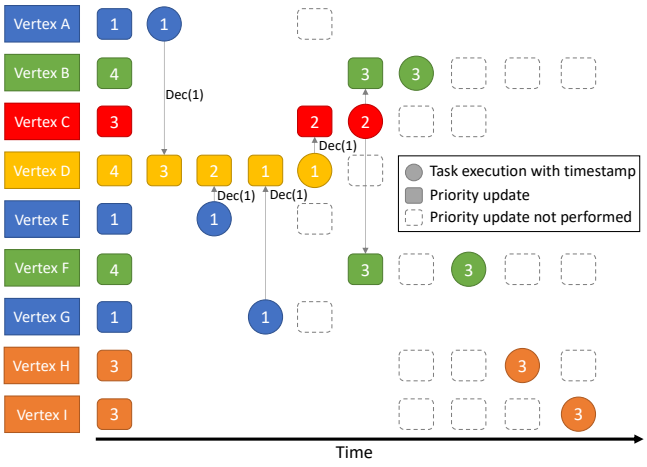


Figure 5: Serial Hive `kcore` execution, showing one task execution per logical time step, corresponding to a dequeue from the modeled priority queue. Tasks execute in priority order, but each task may update the priorities of future tasks, influencing the order in which they are dequeued.

3.3 Programming Hive with patterns

We find that porting a sequential program to the Hive API is straightforward. The bodies of the Hive tasks closely resemble the bodies of the sequential loop iterations. The main challenge of writing a Hive program is recognizing which scheduling *pattern(s)* the algorithm uses, which consequently implies the Hive API calls to use. We observe five patterns across our nine benchmarks.

The UPDATE pattern arises when the main loop of the sequential program unconditionally updates the priority of objects to new values. For example, residual belief propagation [25] (`rbp`) prioritizes each task based on the output of a many-input function and updates the task on every change to any input. Listing 4 shows the sequential loop and Hive task for `rbp`. When the algorithm exhibits the UPDATE pattern, then a call to `update` (line 19) replaces the priority queue's version (line 9).

The CANCEL pattern arises in a surprising context: algorithms that do not need a priority queue for scheduling, but the sequential implementation uses a for loop with known trip count, and iterations may be skipped. For example, greedy maximal independent set (`mis`) finds a subset of vertices in a graph such that no two vertices included in the set are adjacent, and every vertex excluded from the set is adjacent to an included vertex. Listing 5 shows the sequential implementation of `mis`, and how we break it into Hive tasks. Unlike `kcore` and `rbp`, all work, and the order of that work, is known at the start. However, the irregular data dependences between tasks are only known at runtime, making software parallelization more challenging. With the CANCEL pattern, loop iterations will set a flag (line 8) to signal future iterations to exit early (line 5). Hive will instead use `cancel` to remove the equivalent task for such iterations (line 13). Hive is able to implicitly exclude vertices by simply canceling their inclusion, obviating an equivalent to line 8 which explicitly sets `n`'s state to `EXCLUDED`.

The UPDATEMIN pattern manifests in algorithms where task priority represents a cost to be minimized, such as in Dijkstra's algorithm for the single-source shortest path (`sssp`) problem [24, 28]. In this pattern, the sequential code updates the schedule only if doing so

```
1 void sequentialMain() {
2   PriorityQueue pq;
3   for (Message* m : messages)
4     pq.enqueue(m, getNewPrio(m));
5   while (!pq.empty()) {
6     Message* m = pq.dequeueMax();
7     m->val = getNewVal(m);
8     for (Message* n: m->neighbors())
9       pq.update(n, getNewPrio(n));
10 }}
11 void updateMessage(Message* m, Timestamp ts) { // Task
12   m->val = getNewVal(m);
13   for (Message* n: m->neighbors())
14     hive::update(&updateMessage, n, getNewTimestamp(n));
15 }
16 void hiveMain() {
17   hive::init(messages.size());
18   for (Message* m : messages)
19     hive::update(&updateMessage, m, getNewTimestamp(m));
20   hive::run();
21 }
```

Listing 4: Hive UPDATE pattern used to implement `rbp`

```
1 void sequentialMain() {
2   for (Vertex* v : G.V)
3     v->state = UNKNOWN;
4   for (Vertex* v : G.V) {
5     if (v->state == EXCLUDED) continue;
6     v->state = INCLUDED;
7     for (Vertex* n : v->neighbors())
8       n->state = EXCLUDED;
9 }}
10 void include(Vertex* v, Timestamp ts) { // Task
11   v->state = INCLUDED;
12   for (Vertex* n : v->neighbors())
13     hive::cancel(n);
14 }
15 void hiveMain() {
16   hive::init(G.n);
17   for (Vertex* v : G.V) {
18     v->state = EXCLUDED;
19     hive::update(&include, v, v->id);
20   }
21   hive::run();
22 }
```

Listing 5: Hive CANCEL pattern used to implement `mis`

```
1 void sequentialMain() {
2   PriorityQueue pq = {source, 0};
3   while(!pq.empty()) {
4     Vertex* v = pq.dequeueMin();
5     for (Vertex* n: v->neighbors())
6       if (v->dist + dist(v, n) < n->dist) {
7         n->dist = v->dist + dist(v, n);
8         pq.update(n, n->dist);
9 }}}
10 void visitVertex(Vertex* v, Timestamp ts) { // Task
11   for (Vertex* n : v->neighbors())
12     hive::updateMin(&visitVertex, n, ts + dist(v, n));
13 }
14 void hiveMain() {
15   hive::init(G.n);
16   hive::updateMin(&visitVertex, source, 0);
17   hive::run();
18   hive::extract(distances);
19 }
```

Listing 6: Hive UPDATEMIN pattern used to implement `sssp`

lowers the priority value of an object. Conversely, sequential algorithms using `dequeueMax` and monotonically increasing objectives use the same API by inverting priority: subtracting it from a large number. Listing 6 shows in `sssp` that a Hive call to `updateMin` (line 12) replaces both distance tracking and a conditional update in the sequential version (lines 6-8).

The INCREMENTAL pattern is illustrated by `kcore`. Listing 1 and Listing 3 show the sequential and Hive code, respectively. To use the INCREMENTAL pattern, the programmer initially calls `update` to bind a set of tasks to objects, then uses `incrTS` or `decrTS` to adjust their timestamps.

The POSTPONE pattern is a special case of the INCREMENTAL pattern when increments use *strictly positive* deltas: tasks are rescheduled only further in the future, never earlier. When this constraint is met, an algorithm could functionally use either pattern. The POSTPONE pattern trades programming complexity for better performance on applications with a high update-to-dequeue ratio. Listing 7 shows the sequential and Hive implementations of the greedy approximate algorithm for unit-cost set cover [40] (`setcover`). Its objective is to

```
1 void sequentialMain() {
2   PriorityQueue pq;
3   for (Set* s : sets)
4     pq.enqueue(s, s->cardinality);
5   while (!pq.empty()) {
6     Set* s = pq.dequeueMax();
7     if (s->cardinality == 0) break;
8     addToCover(s); // Not shown for brevity
9     for (Elem* e : s->elements) {
10      if (e->state == COVERED) continue;
11      e->state = COVERED;
12      for (Set* s1 : e->sets)
13        pq.decrementPrio(s1);
14 }}}
15 int prios[sets.size()];
16 void coverElem(Elem* e, Timestamp ts) { // Task
17   if (e->state == COVERED) return;
18   e->state = COVERED;
19   for (Set* s1 : e->sets)
20     prios[s1->id]++; // Decrement effective cardinality
21 }
22 void addSet(Timestamp ts, Set* s) { // Task
23   if (prios[s->id] > ts) {
24     hive::enqueue(&addSet, prios[s->id], s);
25     return;
26   }
27   addToCover(s);
28   for (Elem* e : s->elems)
29     hive::updateMin(&coverElem, e, ts);
30 }
31 void hiveMain() {
32   hive::init(elems.size());
33   for (Set* s : sets) {
34     prios[s->id] = MAX - s->cardinality;
35     hive::enqueue(&addSet, prios[s->id], s);
36   }
37   hive::run();
38 }
```

Listing 7: Combining the POSTPONE pattern and UPDATEMIN pattern to implement `setcover`

find the minimum number of sets whose union covers all elements in a universe. Its priority heuristic processes sets in decreasing order of (remaining) cardinality. `setcover` exhibits both the POSTPONE pattern and the UPDATEMIN pattern. The POSTPONE pattern uses `enqueue` instead of `update` or `incrTS` (line 35), and a counter array to accumulate increments (line 20). When a POSTPONE pattern task dequeues, it checks its object's counter and may re-enqueue itself for a later timestamp (lines 23-25). When the algorithm runs many increments per dequeue, the POSTPONE pattern minimizes the number of calls into the Hive API by accumulating updates. This is safe because the strictly positive deltas guarantee that a task will dequeue *before* any task that semantically replaces it, so it is safe to postpone the enqueues.

3.4 Comparison with other models

Hive generalizes upon the Swarm [37] execution model. Swarm tasks *enqueue* timestamp-ordered tasks that have no binding to any object, and therefore must run at their originally assigned timestamps. Hive extends Swarm to also support object-task binding and priority updates. Any Swarm-only program can run on Hive, avoiding object table initialization.

Fractal [72] extends Swarm to enable composition of nested speculative parallelism. One implication is that Fractal can implement a

modeled *unrestricted* [74] priority queue, whereas Swarm models a *monotone* [74] priority queue: a child's timestamp must be greater than or equal to its parent's. Hive is orthogonal to Fractal. While we do not explore the implications of priority updates in the context of nested speculative parallelism in this paper, Hive requires no special handling for Fractal domains and can be implemented on a Fractal system similarly to a Swarm one.

Likewise, it is unlikely that any special handling is required to adapt the hardware mechanisms of Espresso [39] to Hive to coordinate speculative and non-speculative ordered tasks with updates, though we leave verifying this for future work.

4 SPECULATIVE PARALLELISM AMID PRIORITY UPDATES

Hive software directly conveys its dynamic scheduling needs to hardware, expressing implicit parallelism through task enqueues and updates. Our goal with the Hive implementation (Sec. 5) is to extract abundant parallelism among the queued tasks. Since safe parallel executions for these irregular algorithms cannot be statically determined at compile time, Hive executes queued tasks in parallel and out of order, speculating that they are independent. Specifically, Hive speculates that for every executing task:

- its data-dependent predecessors have performed their stores,
- its parent did not misspeculate, and
- it will not become Moot: replaced or canceled.

The Swarm architecture [37] similarly speculates on the first two conditions (data and control dependences, respectively), making it a natural baseline upon which to build Hive. This section gives an overview of Hive's approach to parallelism, identifying similarities between Swarm and Hive, then highlighting the key insights that enable Hive to speculate efficiently amid priority updates.

4.1 Similarities and differences with Swarm

Both Swarm and Hive detect data misspeculation by tracking the memory read and write sets of all tasks and piggy-backing on cache coherence requests. When two tasks access the same data with at least one writing, both systems identify a *dependence order violation* based on task timestamps and access types, and abort and restart the later-ordered task if necessary.

Both systems handle control misspeculation by tracking children tasks. A parent task may have created its children based on incorrect control flow due to misspeculating on data. If the parent aborts, both systems abort all its control-misspeculated descendents, recursively.

The key distinction of Hive hardware from Swarm is its builtin support for updates to the schedule, or how it speculates on *scheduler-carried dependences*. While Swarm can only rely on its support for data and control misspeculation, Hive maintains multiple versions of scheduler-dependent tasks, with all but one per object being in a Moot state. This reduces Moot task overheads, while retaining the ability to recover from priority update misspeculation.

4.2 Scheduler dependences and Moot tasks

We identify the *scheduler-carried dependence* as a new class which resembles both data and control dependences, but is neither. A task t is *scheduler-dependent* on task s if, when s appears to begin executing, t is scheduled after s, and either

```
1 Timestamp prios[G.n]; // Scheduling metadata
2 void removeV(Timestamp ts, Vertex* v) { // Task
3   if (prios[v->id] < ts) return;
4   for (Vertex* ngh : v->neighbors()) {
5     if (prios[ngh->id] <= ts) continue;
6     prios[ngh->id]--;
7     swarm::enqueue(&removeV, prios[ngh->id], ngh);
8   }
9 }
10 void main(int argc, char** argv) {
11   for (Vertex* v : G.V) {
12     prios[v->id] = v->degree;
13     swarm::enqueue(&removeV, prios[v->id], v);
14   }
15   swarm::run();
16   coreness = prios;
17 }
```

Listing 8: The Swarm implementation of `kcore` transforms scheduler-carried dependences into data and control dependences, as did Listing 2 for sequential code.

- s mutates scheduler state corresponding to t, or
- t is scheduler-dependent on u, and u is scheduler-dependent on s.

Scheduler-carried dependences arise in systems where scheduler state of future tasks is accessible and mutable. One example is when a Hive task s cancels the task t bound to some object. The execution or existence of t is scheduler-dependent on s. Another example is in processors with self-modifying code [78], if we view each instruction as an individual task. To exit a loop, a store instruction would overwrite the jump target address in the memory location of a later PC [29]. The jump instruction is scheduler-dependent on the store. Swarm tasks are never scheduler-dependent, because Swarm has no mutable scheduler state.

Like how predication can transform control dependences into data dependences [9], one can transform scheduler-carried dependences into a combination of control and data dependences. Consequently, Swarm can implement the Hive execution model using only its data- and control-dependence speculation. For example, Listing 8 shows a Swarm implementation of `kcore`. We replace each Hive operation with reads and writes of scheduling metadata in memory (`prios`) and task `enqueue` (line 7). Now, every update operation produces a new task, and each task first checks memory to see if it is still scheduled to run, exiting early if not (line 3).

Priority update operations often outnumber dequeues in algorithms with updatable priority queues (Sec. 2). Therefore, the majority of tasks in Swarm implementations will dequeue, perform a single memory read, and exit early with no effect on program state. For example, the early exit path of Listing 8 will trigger more than 36× more often than not (Fig. 3). We call these early exiting tasks Moot because they might as well have not run at all.

In a Swarm system, every Moot task consumes cycles on a core while waiting on its memory access, and its speculative state with non-empty read set consumes precious hardware resources until it commits in order. Wasting core and speculation resources on Moot tasks can cause stalls, hurting program performance.

In contrast, Hive speculates on a lack of scheduler-carried dependences among tasks, but it recovers from misspeculation by holding multiple speculative *task versions* for the same object. Unlike Swarm, Hive does not treat these versions as separate tasks: only one will appear to dequeue and run, matching the sequential

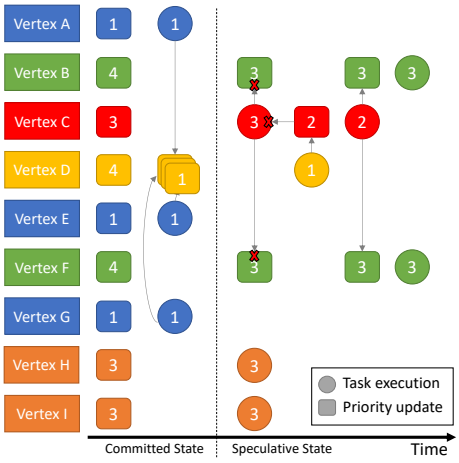


Figure 6: Speculative parallel Hive execution of `kcore`.

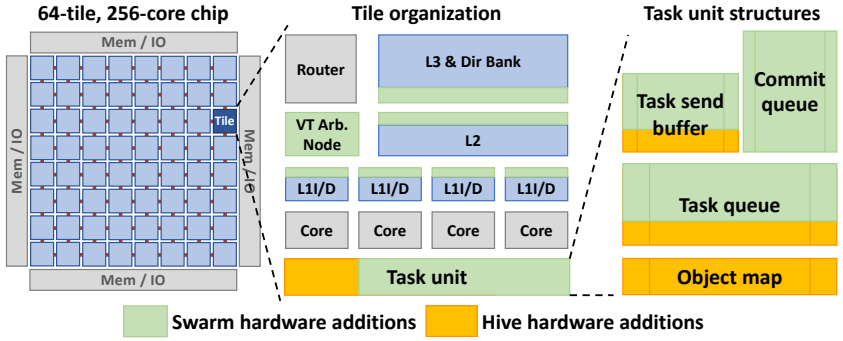


Figure 7: Swarm and Hive system configuration.

semantics (Sec. 3). Hive temporarily holds the others in an explicit Moot state. Hive does not execute Moot task versions and aborts a task if it becomes Moot after dequeue. Hive clears Moot versions out of its hardware resources when their fate becomes non-speculative. This is always earlier than they would be committed as normal tasks in Swarm. We provide more detail in Sec. 5.

4.3 Example speculative execution

Fig. 6 shows a snapshot of the committed and speculative state of a Hive system running `kcore` on our example graph using speculative parallelism. At this moment, the blue vertex tasks have committed, so there exists only one version of the yellow vertex task, which is still speculative. The red vertex has received the speculative update from the yellow vertex task, which has caused its original task at timestamp 3 to become Moot. This Moot task had already begun executing at that time, a misspeculation which has been aborted. The valid version of this task, with timestamp 2, has already begun speculatively executing, and will become the only version when the yellow vertex task commits. This will destroy the version of the red task with timestamp 3 when the yellow task with timestamp 1 commits, freeing speculative resources earlier, and obviating a Swarm-style re-execution of an early exiting Moot task.

5 HIVE IMPLEMENTATION

Given the overview of Hive's approach to speculative parallelism, we now present an implementation of Hive as an extension to the Swarm microarchitecture [36, 37, 39], visualized in Fig. 7. With restrictions, Hive could also be adapted to the Chronos [3] accelerator for speculative ordered parallelism, which we leave to future work. We first describe Swarm's main features for task-level data and control speculation. We then turn to Hive's modifications that enable detection and recovery from scheduler-dependence misspeculation.

5.1 Baseline Swarm microarchitecture

Swarm makes modest changes to a tiled, cache-coherent multicore. Each tile has a cluster of cores, each with its own private L1 cache. Cores in the same 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 extracts parallelism among queued timestamped tasks by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest active task. Swarm efficiently supports fine-grain tasks and a large speculation window through five techniques: hardware task management, large task queues, scalable speculation, high-throughput ordered commits, and locality-aware execution. Hive adapts each of these in support of speculating on scheduler dependences.

Hardware task management: Each task unit queues runnable tasks and stores the speculative state of finished tasks until they commit. A task is represented by a *task descriptor* that contains its function pointer, 64-bit timestamp, arguments, and other metadata.

`swarm::enqueue` creates a task with a `create_task` instruction with arguments passed through registers. The local task unit asynchronously sends the task descriptor to a remote tile. The parent's speculative state tracks where each child is enqueued, so that the task unit can send parent commit or abort notifications. Receipt of a parent abort notification aborts and discards the child, while a parent commit notification permits queue virtualization (see below).

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

Large task queues: Each task unit has three main structures: *(i)* the *task send buffer* (TSB) holds newly created task descriptors to be asynchronously sent to their destination tile, *(ii)* the *task queue* (TQ) holds descriptors for every task received at the tile, and *(iii)* the *commit queue* (CQ) holds the speculative state of tasks that have finished executing but cannot yet commit.

These structures support tens of speculative tasks per core (e.g., 64 TQ entries and 16 CQ entries per core) to implement a large window of speculation (e.g., 16K tasks in the 256-core chip of Fig. 7). Nevertheless, the queues can fill up, requiring some simple actions to ensure forward progress. Specifically, tasks that receive parent-commit notifications can be *spilled* to memory to free TQ entries, virtualizing this structure. If no tasks can be spilled, queue resource exhaustion is handled by either stalling task creation or aborting higher-timestamp tasks to free space.

Scalable data and control speculation: Swarm enhances prior data-dependence speculation mechanisms to support the large number of speculative tasks. Swarm uses eager version management and eager conflict detection using Bloom filters, like LogTM-SE [84]. Swarm forwards still-speculative data to later readers. Swarm detects conflicts at cache-line granularity and leverages the cache hierarchy to substantially reduce the number of conflict checks and their cost. When a task aborts, Swarm sends its children abort notifications and rolls back its memory writes, recursively and selectively aborting only the descendants and data-dependent tasks.

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

Exception model: Espresso [39] defines and implements an exception model for Swarm. Any attempt by a speculative task to perform an irrevocable action (e.g., segmentation fault or system call) triggers a *speculative exception*. The task aborts, releases its core, and is queued in a not runnable, *exceptioned* state. The task transitions to idle and runnable if a conflict is later found on its read set (suggesting it may have misspeculatively triggered the exception), or when it becomes the earliest active task.

Tracked and untracked memory: Swarm tasks typically read and write memory with accesses tracked by hardware-managed speculation. Capsules [39] introduces regions of *untracked* memory for select task code to issue accesses that bypass hardware-managed speculation. Akin to virtual memory protections, speculative task accesses to untracked memory trigger a speculative exception, protecting these regions from misspeculating tasks that lose integrity.

Ordered commits: Swarm adapts the virtual time algorithm [35] to achieve high commit throughput. Tiles periodically communicate with an arbiter (e.g., every 200 cycles) to determine the VT of the earliest (lowest-VT) active (unfinished) task in the system. All tasks with lower VTs can then commit. This scheme uses a hierarchical min reduction and can commit many tasks per cycle, scaling to hundreds of cores with tasks as short as a few instructions.

Locality-aware execution: Swarm is flexible in where to send a task to be queued. To exploit data locality on fine-grain tasks, it leverages a technique called *spatial hints* [36]. A hint is an optional integer that abstractly denotes the data a new task is likely to access (e.g., a vertex ID). When a core creates a task, the task unit hashes its 64-bit hint to determine the destination tile ID. The hint is stored in the task descriptor. The task unit sends tasks without hints to random tiles. Thus, Swarm runs same-hint tasks at the same tile.

5.2 Hive microarchitecture

Hive generalizes the Swarm microarchitecture to support *(i)* logically binding ordered tasks and objects, and *(ii)* speculating on the outcome of scheduler-carried dependences. To implement the former, Hive introduces the *object table* in memory. To achieve the latter, Hive adapts Swarm task unit structures, shown in Fig. 7, to enable *task versioning*. We first describe Hive hardware *assuming only support for the* `hive::update` operation, along with `hive::init`. We then describe support for `getTS`, `updateMin`, `incrTS`, `decrTS`, and `cancel`. As it is object-agnostic, `hive::enqueue` uses identical hardware features as `swarm::enqueue`.

Object table entries represent non-speculative object-task bindings. The object table has an entry for every Hive object o, holding a unique identifier (UID) for the task version currently (or last) bound to o. This task UID is opaque to software and can be cheaply derived from a mechanism similar to the Swarm VT tiebreaker.

Misspeculating tasks must never corrupt the object table, so we allocate it in untracked memory (Sec. 5.1). Tasks never read or write the object table directly. Instead, task units non-speculatively read and write UIDs in the object table to facilitate TQ virtualization. They update object-task bindings when the creation of a new task version becomes non-speculative (see below).

The programmer specifies the number of object table entries to allocate with `hive::init`. We implement the object table as an array and leave dynamic creation and destruction of objects to future work (e.g., via hash table or tree). The Hive software runtime initializes all entries to a task UID signifying no task (e.g., 0).

Speculative task versioning: Hive buffers task descriptors in the TQs for all task versions for all objects, alongside any `enqueued` tasks unbounded from objects. A task calling `update` creates a task with a similar asynchronous send scheme as `enqueue`. Like `enqueue`, the parent tracks the location of its child. Unlike `enqueue`, the child may be speculatively replacing (or replaced by) another task for the same object, making it a speculative task version.

Hive sends new task versions for the same object to the same tile. Like Swarm's spatial hints insight, tasks bound to the same object are likely to access the same data, leveraging locality. Unlike (optional) spatial hints, co-locating same-object tasks is required to avoid races on the object table and simplify detection of scheduler-dependence misspeculation.

Mootness detection and recovery: Hive must detect and recover from two cases of scheduler-dependence misspeculation. First, a task version that is idle, running, or finished could be replaced by an update operation, making it speculatively Moot. Second, a task version previously found to be Moot may have been replaced by a misspeculating task calling `update`, and should be restored.

Fig. 8 shows how Hive expands the Swarm task descriptor to facilitate mootness detection. Update operations add the VT of the task responsible for creating the task version. This is usually the parent that directly called `update`, but read-only spawner trees [86] propagate the `parentVT` from their root task to increase parallelism. Hive replaces a 16-bit spatial hint hash with a 64-bit full object ID. `enqueued` tasks omit the `parentVT` and object ID, as in Swarm. Hive adds three flag bits to encode the type of operation that created this task version (e.g. `update`, `enqueue`, etc.)

A task unit detects which task versions are newly made Moot when it receives a new task descriptor for a priority update to object o. The task unit inserts the incoming task version t_i to a free slot in the TQ and compares two VT fields of t_i with those of all task

Figure 8: Task descriptor for Swarm (top) and Hive (bottom).

versions for o in the TQ. For every queued task version, t_q:

$$parentVT(t_q) < parentVT(t_i) < VT(t_q) \Rightarrow t_q \text{ is Moot}$$
$$parentVT(t_q) < VT(t_q) < parentVT(t_i) \Rightarrow \text{neither is Moot}$$
$$parentVT(t_i) < parentVT(t_q) < VT(t_i) \Rightarrow t_i \text{ is Moot}$$
$$parentVT(t_i) < VT(t_i) < parentVT(t_q) \Rightarrow \text{neither is Moot}$$

If task t is idle, $VT(t)$ is its timestamp appended with all 1s. The first case implies t_i replaces t_q. The second implies that both tasks will appear to run (unless another priority update for o intervenes). In an equivalent sequential execution, t_q runs, o is then taskless, t_i then binds to o, and finally t_i runs. The third and fourth cases swap t_q and t_i. When Hive detects that a task version is newly Moot, it aborts the task (if running or finished) and queues it in a not runnable, Moot state. Moot versions are not considered active.

A task unit detects which tasks in Moot state should have been runnable when it receives a parent abort notification for a local child task version, t_a, of object o. As in Swarm, the task unit aborts t_a. Unlike Swarm, it also performs the same mooting comparisons as above, replacing t_i with t_a, to reconsider the previously Moot tasks. All versions that were made Moot due to t_a repeat the VT test as if they were newly received, possibly concluding that they should be idle instead of Moot. No task version ever becomes newly Moot due to receipt of an abort notification.

Object map: Hive adds an *object map* to every task unit to accelerate task-version queries. This associative array maps an object ID to the set of TQ entries that hold task versions for the given object. We move the object ID field out of the TQ and into the object map.

Fig. 9 illustrates an object map and TQ being queried for an incoming update. This task unit is holding a mix of `ENQUEUE`- and `UPDATE`-flagged tasks and task versions. Every color represents a distinct task, with gray showing tasks unbounded from objects. Several speculative task versions are entries with the same color. Arrows are illustrative only, but connect a task version to the others it renders Moot. There are two unbounded tasks and three tasks speculatively bound to two objects. Green and blue share one object, being distinct tasks instead of versions of the same task. Both will appear to run sequentially (see above). In a sequential priority queue, green and blue would not occupy the queue at the same time, but task versioning enables Hive to uncover speculative parallelism.

Clearing Moot task versions: When a task commits, its children are no longer control-speculative but they may remain data-dependent or scheduler-dependent on other speculative tasks. Swarm and Hive virtualize the TQ by making idle control-non-speculative tasks

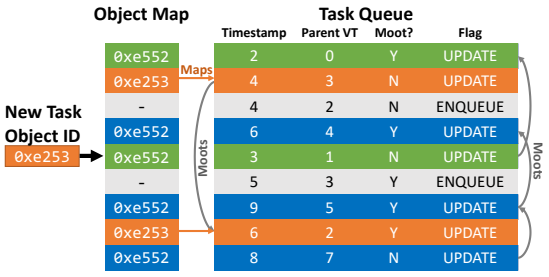


Figure 9: Querying the object map and task queue to find the task versions of an object. `ENQUEUE`-flagged tasks have a null entry in the object map.

spillable to memory (Sec. 5.1). Hive also clears Moot tasks from the TQ and object map by exploiting scheduler-non-speculative object-task bindings. Given a parent commit notification for local child task t, a task unit *(i)* discards all task versions rendered Moot by t, given the mooting comparisons seen earlier, and *(ii)* writes t's UID to the object table. A task unit also discards a task version, t, when it is restored from a spill to memory and the UID in the object table no longer matches its own, i.e., t was non-speculatively replaced while spilled. Whereas Moot Swarm tasks execute and wait in the commit queues, Hive clears Moot task versions as soon as possible.

Supporting other operations: Unconditional `hive::update` easily leverages Hive's scheduler-dependence speculation described so far. The other Hive operations require some additions.

`getTS, incrTS, and decrTS:` Timestamp-relative Hive operations incur data dependences among tasks on the timestamps of tasks bound to objects. Hive optionally augments the object table with an array of timestamps in *tracked* memory to enable data-dependence speculation, only if these operations are needed. Each call to `update`, `incrTS`, or `decrTS` speculatively writes an entry of this timestamp array (with remote tasks to exploit locality). `getTS` reads it. Together, they enable speculative forwarding of object timestamps.

`updateMin:` Although `updateMin` could naively call `getTS`, its semantics match an *ordered put* [55, 68] operation, which is a qualified write and *not* a read. Hive leverages `updateMin` semantics to eliminate the speculative timestamp array for its objects. `updateMin` flags its child task with `UPDATEMIN` in the task descriptor, and Hive uses a different mooting comparison for these tasks:

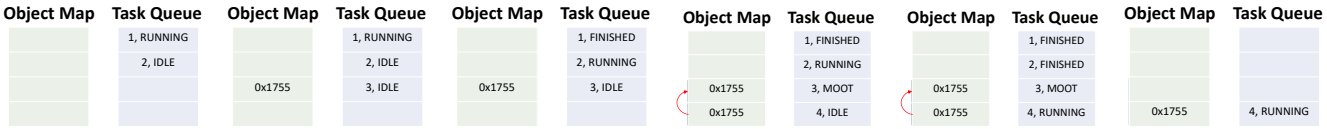
$$VT(t_i) < VT(t_q) \Rightarrow t_q \text{ is Moot}$$

To clear Moot versions for `UPDATEMIN`-flagged tasks, Hive replaces the UID written to the non-speculative object table with the timestamp of a child receiving a parent commit notification. When a task unit receives a new `UPDATEMIN` task version, the task unit reads the non-speculative object table timestamp, and discards the incoming task if its timestamp is higher. This enables an `updateMin` task to keep mooting later task versions long after commits.

`init` is a variant of `malloc` that allocates from untracked memory (and optionally regular tracked memory). It also accepts flag parameters that indicate whether the programmer will use `update` or `updateMin` on the objects, and whether they require `getTS`. `update` and `updateMin` cannot be used on the same objects because they use the object table differently. Supporting `getTS` allocates another array and incurs additional data dependences through tracked memory. Most of our benchmarks do not use `getTS`, `incrTS`, nor `decrTS`. None of our benchmarks need both `update` and `updateMin` for the same object. The program may call `init` multiple times to initialize different sets of objects with different flags.

`cancel` is syntactic sugar around `update` or `updateMin` (Table 1). A `cancel`'s child task version is empty and has timestamp zero or infinity, depending on the object's `init` configuration. When this NOP task version receives a parent commit notification, the task unit discards the child along with all versions it caused to be Moot.

All `cancel` child task versions obey the mooting comparison for their object configuration. Therefore, calling `cancel` on an object initialized for `updateMin` can be cheaper than on an object initialized for `update`. This is because earlier calls to `cancel` will cause



a) Initially, task t_1 is running b) t_1 updates object 0x1755 with task t_3 c) t_1 finishes; t_2 starts running d) t_2 updates object 0x1755 to t_4 e) t_2 finishes; t_4 starts running f) t_1 and t_2 commit; MOOT t_3 is discarded

Figure 10: Lifetime of a priority update in the object map and task queue. Task t_i has timestamp i.

Table 2: Sizes and estimated areas of task unit storage elements for Swarm (in gray) and Hive (in black).

		Task Queue	Task Send Buffer	Commit Queue filters (2–port)	Commit Queue other	Order Queue (TCAM)	Object Map (CAM)
Entries		256	96	64	64	256	256
Entry Size (bytes)	S	51	45	16×32	36	2×8	N/A
	H	65	67	16×32	36	2×8	8
Size (KB)	S	12.75	4.22	32	2.25	4	0
	H	16.25	6.28	32	2.25	4	2
Est. area (mm^2)	S	0.032	0.016	0.149	0.009	0.175	0
	H	0.043	0.028	0.149	0.009	0.175	0.011

later calls to be MOOT for UPDATEMIN-flagged cancel, but not for UPDATE-flagged cancel. Consequently, programs with the CANCEL pattern should initialize their objects for updateMin if application semantics allow it. We use this approach in our benchmarks.

extract is a memcpy from the object table. Because it may access untracked memory it may not be called by speculative tasks.

5.3 Priority updates example

Fig. 10 illustrates the lifetime of priority updates on an object in a 1-core Hive system. We use a total order of timestamps to hide VT details. Initially, the TQ holds two tasks with timestamps 1 and 2. Each task makes an update on object 0x1755 with timestamps 3 and 4, respectively. The update with timestamp 4 replaces the update with timestamp 3, so when the tasks with timestamps 1 and 2 are finished, the task starts running with timestamp 4, because the version with timestamp 3 is MOOT. When the task with timestamp 2 commits, this discards the version with timestamp 3.

5.4 Hive overheads

Swarm adds modest overheads [37] to a multicore and Hive adds more, with the key addition of a 2KB CAM for the object map per task unit. Table 2 breaks down the hardware storage overhead for both systems. We use CACTI 7.0 [13] to estimate the area of Bloom filters, SRAMs, and CAMs for a 32nm process. We estimate the order queue area by scaling a commercial 28nm TCAM [11]. The total storage area of a Hive task unit is less than 3% of a 45nm Nehalem processor [26] when scaled up to a 45nm process.

Hive increases the memory footprint by adding a 64-bit word in untracked memory for every object, and possibly another in tracked memory when getTS/incrTS/decrTS are enabled. However, this increase can be offset by replacing an identical array the program would have allocated anyway. For instance, in sssp and bfs, the object table replaces the programmer-allocated array of vertex distances, producing no net change in memory requirements.

Table 3: Configuration of the 256-core system.

Cores	256 cores in 64 tiles (4 cores/tile), 2 GHz, x86-64 ISA; single-issue in-order, scoreboarded (stall-on-use) [36, 39]
L1 caches	32 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	1 MB, per-tile, 8-way, inclusive, 9-cycle latency
L3 cache	256 MB, shared, static NUCA [41] (4 MB bank/tile), 16-way, inclusive, 12-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	Four 8×8 meshes, 192-bit links, X-Y routing, 1 cycle/hop when going straight, 2 cycles on turns (like Tile64 [77])
Main mem	4 controllers at chip edges, 120-cycle latency
Queues	64 task queue (and object map) entries/core (16384 total), 24 task send buffer entries/core (6144 total), 16 commit queue entries/core (4096 total)
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [17] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Virtual time	128-bit virtual times, tiles send updates to virtual time arbiter every 200 cycles
Spills	Spill 15 tasks when the task queue is 85% full
Task mapper	Statically hash object IDs and spatial hints to tiles [36]

6 EVALUATION

We evaluate Hive across nine benchmarks that require, or benefit from, priority scheduling with updates. We find that Hive consistently outperforms Swarm and software-only parallel implementations, with speedups of up to 2.8× over Swarm at 256 cores (gmean 52%) and more over software. We then characterize Hive and Swarm performance sensitivity to graph structure, and finally tease apart the performance impact of restricting scheduling features.

6.1 Methodology

Modeled system: We adapt an open-source,[2] cycle-level, execution-driven simulator based on Pin [47, 57] to model Hive, Swarm, and multicore systems of up to 256 cores, shown in Fig. 7, with parameters in Table 3. Swarm parameters are consistent with prior work [36, 37, 39, 72, 86]. We use detailed core, cache, network, and main memory models, and simulate all task and speculation overheads (e.g., task traffic, running misspeculating tasks until they abort, simulating conflict and mootness check and rollback delays and traffic, etc.). We also simulate smaller systems with square meshes ($K \times K$ tiles for $K \leq 8$), keeping per-core cache sizes and queue capacities constant. Since aggregate cache and queue capacity grows, we see superlinear speedup on several benchmarks.

Benchmarks: Table 4 details the benchmarks we use to evaluate Hive. It includes their provenance, inputs, and characteristics at 1 core: total run time for tuned Swarm, average task length, and performance vs. tuned serial implementations. It also summarizes

[2] https://github.com/SwarmArch/sim

Table 4: Benchmarks: sources and inputs; run time, task length, and tuned serial-relative performance on a single-core system.

Benchmark	Input	SW parallel strategy	Hive programming pattern(s) (Sec. 3.3)	1-core cycles	Avg. task length		1-core vs. serial	
					Swarm	Hive	Swarm	Hive
kcore [23]	com-Orkut [82]	Bulk-Synchronous	Incremental	219B	364	241	0.42×	1.09×
setcover [23]	com-Orkut [82]	Relaxed PQ	UpdateMin and Postpone	14.6B	137	140	2.47×	1.53×
astar [37]	Germany roads [2]	Relaxed PQ [51, 56]	UpdateMin	1.4B	848	1056	1.22×	1.36×
bfs [45]	hugetric-00020 [12, 22]	Bulk-Synchronous	UpdateMin	3.2B	117	150	0.76×	0.96×
sssp [58]	East USA roads [1]	Relaxed PQ [51, 56]	UpdateMin	2.0B	246	258	1.65×	2.24×
msf [69]	kronecker_log16n[12, 22]	Speculation [14]	UpdateMin and Cancel	0.50B	137	257	2.24×	3.08×
mis [69]	R-MAT [19]	Speculation [14, 15]	Cancel	2.0B	119	81	0.68×	0.98×
mm [69]	com-Orkut [82]	Speculation [14, 15]	Cancel	22.5B	147	147	0.53×	0.51×
rbp [4]	200×200 Ising [20]	Relaxed PQ [64]	Update	18.4B	1128	1697	0.90×	1.23×

the strategies of the software-only parallel implementations, and the Hive programming pattern(s) we leveraged (Sec. 3.3).

We ported four graph benchmarks to Hive (`astar`, `bfs`, `sssp`, and `mis`) from prior Swarm work [37, 39, 72]. We exclude those Swarm benchmarks that do not use object-based priority updates, such as discrete-event circuit and architectural simulation, as Swarm and Hive are equivalent. `mis` (Sec. 3.3) does not use dynamic scheduling but benefits from the Hive CANCEL pattern. We leave investigation of more algorithms with the CANCEL pattern to future work. `astar` (A∗ pathfinding [31]), `bfs` (breadth-first search), and `sssp` (Sec. 3.3) can be expressed with priority updates. Software-parallel `astar` uses the same relaxed implementation as in Chronos [3].

We also ported one statistical inference (`rbp`), one optimization (`setcover`), and three graph (`kcore`, `msf`, `mm`) algorithms to Hive and Swarm. `kcore` (Sec. 2) and `setcover` (Sec. 3.3) use the same graph data structures as the software-parallel Julienne [23], but obviate its bucket-based scheduler. Similarly, `msf` (minimum spanning forest) and `mm` (greedy maximal matching) use the graph structures of PBBS [69]. We use the bipartite double cover of our input graph as input to `setcover`, like Julienne. Hive, Swarm, and serial `msf` implement Prim's algorithm [61], while the software-parallel version implements Kruskal's algorithm [43]. We ported serial and parallel `rbp` (Sec. 3.3) from Java [4] to C++, then ported to Hive and Swarm.

All Swarm versions, except `mis` and `mm`, add a software-managed scheduling metadata array and early exiting conditions at the top of each task. In some cases, (`kcore`, `astar`, `bfs`, `sssp`) this scheduling metadata is the actual output of the algorithm.

Unless stated otherwise, we report speedups relative to *tuned* 1-core Swarm implementations. Due to hardware task management, 1-core Swarm versions are competitive with (and often faster than) tuned software-only serial implementations, as shown in Table 4. Notable exceptions are `mis` and `mm`, whose serial versions avoid overheads of dynamic scheduling, and `bfs`, which uses a constant-time FIFO queue. The `setcover` Julienne bucket queue at 1 thread is our serial baseline, as it is faster than a Fibonacci heap.

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

6.2 Hive performance

Fig. 11 compares the performance of Hive, Swarm, and software-only parallel versions of our benchmarks, as the system scales from 1–256 cores. Hive always outperforms both, with speedups over parallel software of 3.3× (`rbp`) to 124× (`sssp`) at 256 cores (gmean 23×). Software-only versions struggle to scale to hundreds of cores

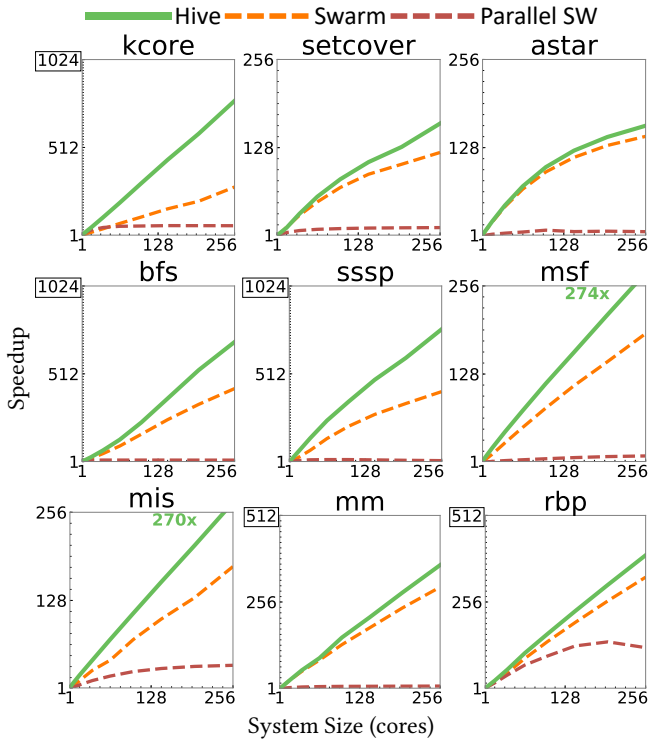


Figure 11: Speedup of Hive, Swarm, and software-only versions on 1–256 cores, normalized to tuned 1c Swarm.

on these inputs, so we do not consider them further. The benefits of Hive over Swarm vary with algorithm and input. At 256 cores, Hive yields between modest speedups of 11–22% (`astar`, `mm`, `rbp`) to large speedups of 1.9× (`sssp`) and 2.8× (`kcore`) (gmean 52%).

Fig. 12 gives more insight into these results by showing execution time breakdowns for Swarm and Hive versions at 256 cores. Each pair of bars shows a benchmark, with the height of a bar giving the execution time relative to Swarm. Each bar breaks down how cores spend their cycles, executing *(i)* tasks that eventually commit or *(ii)* later abort; and cycles spent *(iii)* spilling tasks to/from memory; *(iv)* stalled on a full TSB or CQ; or *(v)* idle because there are no (non-MOOT in Hive) tasks available to run. The figure overlays the update-to-dequeue ratio of Fig. 3 on the right y-axis. We analyze overall trends first, then focus on outliers `kcore` and `mm`.

Hive reduces total committed task cycles across all benchmarks, but to varying degrees. This typically corresponds to Hive's ability

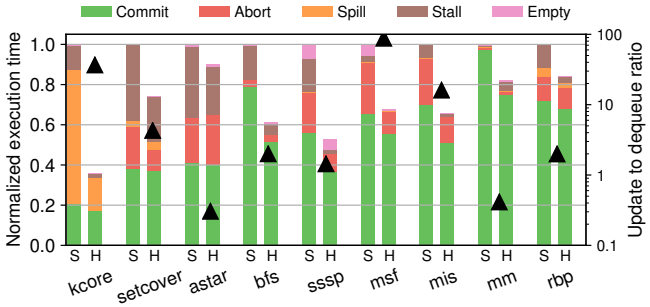


Figure 12: Breakdown of total core cycles at 256 cores, comparing Swarm and Hive. Lower is better.

to avoid useless work: Hive does not execute and commit Moot task versions, whereas Swarm executes Moot tasks with early exits. When tasks are short (Table 4) early exiters can increase committed cycles by over 38% (`bfs`, `sssp`, `mis`). However, when Moot tasks are less common (`setcover`, `astar`) and/or tasks are longer (`astar`, `rbp`), Swarm's increase in committed cycles is meager: less than 6%. Both Hive and Swarm versions of `setcover` benefit from the POSTPONE pattern on the `Set` objects, as it is independent of Hive hardware. This eliminates many Moot tasks in both systems. `astar`'s targeted and directed search of the road graph rarely revisits a vertex, leading to few Moot tasks in either system.

Hive reduces the cycles stalled on resource exhaustion for all benchmarks except `mm`. Hive Moot task versions do not waste precious CQ space, but Swarm Moot tasks do. Moreover, Hive clears many Moot tasks from the TQs at parent commit time, whereas Swarm waits to commit Moot tasks in order. CQ exhaustion is solved both by stalls and aborts, so these gains also translate to some reduced aborts. Hive cuts over 68% (up to 99%) of stall cycles in benchmarks with many Moot tasks (`bfs`, `sssp`, `msf`, `mis`). Similar to committed cycles, Hive cuts fewer stall cycles when Moot tasks are rare and/or tasks are long (`setcover`, `astar`, `rbp`). Hive's `setcover` speedup comes solely from more efficient CQ utilization on its `Element` objects, using the UPDATEMIN pattern.

Hive reduces aborted task cycles not only by curtailing resource exhaustion, but also through *early* mootness detection. Tasks in `setcover`, `sssp`, `msf`, and `mis` can have several children (and recursively, descendants) spanning many distinct timestamps. Hive aborts Moot versions as soon as a new task version *arrives* at a tile, whereas Swarm aborts Moot tasks later when the new task *runs*. Hive is able to clear the tree of descendants earlier, so fewer cycles are spend misspeculating that these tasks will run.

`kcore` stands out for its significant cycles (two thirds in Swarm) spent spilling/filling tasks due to task queue exhaustion. It exhibits an adversarial pattern for Swarm TQs. `kcore` decrements every vertex's effective degree (timestamp), which requires a newly enqueued task each time in Swarm. Since a vertex tasks' timestamps monotonically decrease, they are enqueued in reverse order of dequeue. Swarm TQs fill up with later-ordered Moot tasks, which spill to memory, then refill and execute with an early exit. In contrast, Hive clears most of the Moot task versions *before* they spill to memory and clears the rest when they are restored. Hive considerably reduces cycles wasted on spills.

`mm` is an outlier for its task structure. `mm` is related to `mis` [15]: it finds a subset of edges such that no two edges included in the set share an endpoint, and every edge excluded from the set shares an endpoint with an included edge. To process an edge, each loop iteration reads and writes two vertices. We expected a fine-grain (FG) [36] `mm` Swarm version that accesses the two vertices in two tiny locality-exploiting tasks to outperform a coarse-grain (CG) version with one task per iteration. However, FG tasks increase queue pressure, and unlike `mis`, CG `mm` tasks are almost as short as FG. Queue pressure overwhelms the benefits of locality for FG Swarm, so we report results for CG Swarm `mm`. In contrast, Hive eliminates Moot tasks, so it both reduces queue pressure and exploits the locality of FG tasks, outperforming Swarm by 22%.

6.3 Sensitivity to input graph structure

The performance benefits of Hive depend not only on algorithm semantics but also input graph structure. Fig. 13 shows the cycle breakdowns for `kcore` across three graphs with characteristics summarized in Table 5. Hive and Swarm exhibit similar trends to those in Sec. 6.2 but the profiles differ with graph structure. Hive significantly reduces the cycles spent spilling tasks to memory. This yields over 2× speedup on the social graphs because spills dominate execution time (`orkut`, `lj`), but only 33% speedup on `road`.

Tasks tend to be spilled to memory when they are scheduled for far in the future. In `kcore`, this happens when a low-coreness vertex neighbors a high-coreness vertex. `road` vertices have coreness (nearly) equal to their degree, so their priority is rarely decremented, if at all, as shown by its low update-to-dequeue ratio. The social graphs have high ratios, implying abundant priority decrements. However, `orkut`'s spill cycles do not increase proportionally to the ratio: we observe that its spread of coreness among vertices and fraction of low-degree vertices are lower than in `lj`. Although the ratio of updates to dequeues is one signal that influences the performance benefit of Hive over Swarm, the interaction between graph structure and algorithm ultimately dominates, making the actual speedup hard to predict.

Table 5: `kcore` input graph characteristics.

Input graph	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max core
com-Orkut (`orkut`) [82]	3.1 M	117 M	38	253
LiveJournal (`lj`) [48]	4.8 M	69 M	14	512
East USA roads (`road`) [1]	3.6 M	8.8 M	2.4	6

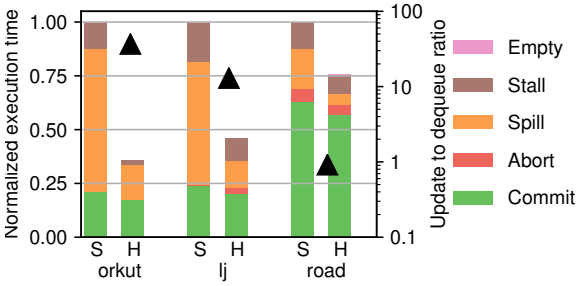


Figure 13: Breakdown of total core cycles at 256 cores for `kcore` on different graphs, comparing Swarm and Hive.

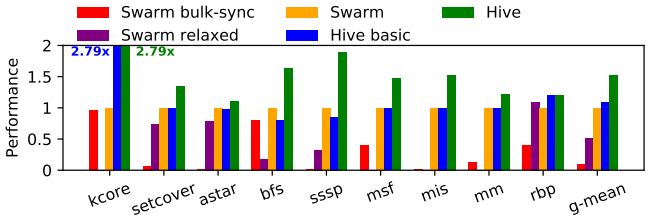


Figure 14: Performance of alternate designs vs. Swarm.

6.4 Contribution of architectural features

Hive and Swarm model strict sequential priority queues and extract parallelism by speculating within and across timestamps. Fig. 14 teases apart the contribution of prioritization features, by comparing performance normalized to Swarm at 256 cores. Bulk-synchronous Swarm executes equal-timestamp tasks speculatively in parallel, but one timestamp at a time. Relaxed Swarm dequeues tasks in tile-local order, but commits them in any order (atomicity only). We omit Relaxed Swarm for algorithms that need a strict schedule (`kcore`, `msf`, `mis`, `mm`). Basic Hive supports the `UPDATE` flag (`update`, `incrTS`, `decrTS`, `getTS` do not change) but not the `UPDATEMIN` flag (`updateMin` instead calls `getTS` and possibly `update`). Bulk-sync Swarm underperforms Swarm slightly (`kcore`, `bfs`) or significantly (all others) due to limited parallelism per timestamp (gmean 0.089×). Relaxed Swarm has abundant parallelism but cores do more overall work, which hurts performance except for `rbp` (gmean 0.51×). Basic Hive is identical to Hive for benchmarks that are implemented without the `UPDATEMIN` flag (`kcore`, `rbp`), and is identical to Swarm for benchmarks that do call `getTS`, but will only call `update` once (`setcover`, `mis`, `mm`). For the remaining benchmarks (`astar`, `bfs`, `sssp`, `msf`), Basic Hive performs slightly worse than Swarm (gmean 0.90×), which suggests that poor use of the Hive API is worse than a well-written Swarm program. Using the wrong Hive pattern destroys data locality and imposes additional data dependences among tasks, outweighing Hive's advantages.

7 ADDITIONAL RELATED WORK

Priority scheduling has a long history in sequential algorithms [24, 28, 31, 40, 61, 76, 79]. Recent work has developed software techniques to scale priority schedulers across cores, and hardware to accelerate the overheads.

Ordered software: Strict parallel scheduling is supported with synchronous bucketing in Julienne [23] and GraphIt [88], speculation in Galois [33, 44] and deterministic reservations [14], and kinetic dependence graphs [34]. These systems perform well when there are ample tasks per priority, or when tasks are large enough to amortize overheads. Hive hardware eliminates these restrictions, unlocking fine-grain ordered parallelism to hundreds of cores.

Relaxing the scheduler's task dispatch order exposes more parallelism and can reduce overheads. Relaxed priority schedulers include OBIM [46, 56], Spraylist [7], Multiqueue [6, 60, 64], PMOD [85], and others [66, 80, 89]. Alternatively, some parallel algorithms coarsen priority values to expose more parallelism for bulk-synchronous schedulers [16, 32, 51]. Relaxation trades off priority drift [66] (and sometimes work efficiency) in exchange for greater scalability. Because they drop strict semantics, these techniques are only applied for algorithms that are resilient to priority inversion.

Hive's `updateMin` has similar semantics to the *write-with-min* operation [68]. However, the latter is implemented with a read and a CAS, suffering data movement under contention, whereas the former sends a task to the object's hosting tile, exploiting locality.

Ordered hardware: Swarm [37] and Chronos [3] abstract a sequential priority queue, targeting multicore and accelerator architectures, respectively. Both speculate on data and control dependences to extract task-level parallelism within and across priorities. However, Swarm detects data misspeculation through coherence and task read and write sets, whereas Chronos associates each task with an abstract object, similarly to Hive, and treats same-object tasks as dependent. Unlike Chronos, Hive allows tasks to read and write any tracked memory. Unlike Hive, Chronos can queue multiple tasks per object simultaneously, but like Swarm, Chronos only supports priority updates with early exiting tasks.

PolyGraph [21] is a graph accelerator that either provides bulk-synchronous strict priority scheduling or relaxed priority scheduling with priority updates. Like Hive's `updateMin`, PolyGraph coalesces updates for the same vertex by keeping the higher priority update. Polygraph accelerates overheads of its software analogs, but similarly requires abundant tasks per priority or algorithms that tolerate priority inversion; it does not speculate across priorities.

Task-based dataflow architectures such as Task Superscalar [27], Picos [73, 83], TDM [18], and Phentos [52] accelerate inter-task dependence analysis and scheduling to find non-trivial parallel schedules without speculation overheads. However, their programming models cannot convey priority-ordered scheduling among tasks, such as enqueuing a task to be executed far in the future.

8 CONCLUSION

Foundational and emerging algorithms depend on a strict task ordering for correctness. However hardware systems that support task order lack crucial update operations. We have examined the semantics of the priority update operation, and in doing so, uncovered a new class of dependence, the *scheduler-carried dependence*. We have described how this new dependence occurs and the necessary invariants required to avoid violating it, as well as implications of these invariants. Using these insights, we designed Hive, an execution model and hardware architecture that implements a scalable priority queue with priority update operations. Hive achieves up to 2.8× speedup over Swarm at 256 cores, while software solutions fail to scale to such large system sizes.

ACKNOWLEDGMENTS

We sincerely thank Milind Kulkarni (our shepherd), Javad Abdi, Isidor R. Brkić, Jerry X. He, Heng Liao, Jing Xia, Victor A. Ying, Xiping Zhou, and the anonymous reviewers for their helpful feedback. We thank Hyun Ryong (Ryan) Lee for his implementations of bulk-synchronous and relaxed-order Swarm. This work was supported in part by Compute Canada, the University of Toronto, NSERC, a Queen Elizabeth II Graduate Scholarship in Science and Technology, and the Engineering Science Research Opportunities Program.

REFERENCES

[1] 2006. 9th DIMACS Implementation Challenge: Shortest Paths.

[2] 2015. OpenStreetMap. https://www.openstreetmap.org

[3] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*. ACM, 1247–1262. https://doi.org/10.1145/3373376.3378454

[4] Vitalii Aksenov, Dan Alistarh, and Janne H. Korhonen. 2020. Scalable Belief Propagation via Relaxed Scheduling. In *Proc. of the International Conference on Neural Information Processing Systems (NeurIPS)*. MIT Press, 22361–22372.

[5] Dan Alistarh, Trevor Brown, Justin Kopinsky, and Giorgi Nadiradze. 2018. Relaxed Schedulers Can Efficiently Parallelize Iterative Algorithms. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 377–386. https://doi.org/10.1145/3212734.3212756

[6] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. 2017. The Power of Choice in Priority Scheduling. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)*. ACM, 283–292. https://doi.org/10.1145/3087801.3087810

[7] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 11–20. https://doi.org/10.1145/2688500.2688523

[8] Dan Alistarh, Giorgi Nadiradze, and Nikita Koval. 2019. Efficiency Guarantees for Parallel Incremental Algorithms under Relaxed Schedulers. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 145–154. https://doi.org/10.1145/3323165.3323201

[9] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proc. of the Symposium on Principles of Programming Languages (POPL)*. ACM, 177–189. https://doi.org/10.1145/567067.567085

[10] J. Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. 2005. Large Scale Networks Fingerprinting and Visualization Using the K-Core Decomposition. In *Proc. of the International Conference on Neural Information Processing Systems (NeurIPS)*. MIT Press, 41–50.

[11] Analog Bits 2011. *4096 x 128 ternary CAM datasheet (28nm)*. Analog Bits. http://mail.analogbits.com/pdf/28nm_TCAM_Product_Brief.pdf

[12] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner (Eds.). 2012. *10th DIMACS Implementation Challenge Workshop*.

[13] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Computer Architecture and Compiler Optimizations (TACO)* 14, 2 (2017). https://doi.org/10.1145/3085572

[14] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 181–192. https://doi.org/10.1145/2145816.2145840

[15] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy Sequential Maximal Independent Set and Matching are Parallel on Average. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 308–317. https://doi.org/10.1145/2312005.2312058

[16] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 443–454. https://doi.org/10.1145/2935764.2935765

[17] J. Lawrence Carter and Mark Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143–154. https://doi.org/10.1016/0022-0000(79)90044-8

[18] E. Castillo, L. Alvarez, M. Moreto, M. Casas, E. Vallejo, J. L. Bosque, R. Beivide, and M. Valero. 2018. Architectural Support for Task Dependence Management with Flexible Software Scheduling. In *Proc. of the International Symposium on High Performance Computer Architecture (HPCA-24)*. IEEE, 283–295. https://doi.org/10.1109/HPCA.2018.00033

[19] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proc. of the International Conference on Data Mining (SDM)*. SIAM, 442–446. https://doi.org/10.1137/1.9781611972740.43

[20] Barry A. Cipra. 1987. An Introduction to the Ising Model. *The American Mathematical Monthly* 94, 10 (1987), 937–959. https://doi.org/10.1080/00029890.1987.12000742

[21] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *Proc. of the International Symposium on Computer Architecture (ISCA-48)*. ACM/IEEE, 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[22] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

[23] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 293–304. https://doi.org/10.1145/3087556.3087580

[24] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271. https://doi.org/10.1007/BF01386390

[25] Gal Elidan, Ian McGraw, and Daphne Koller. 2006. Residual Belief Propagation: Informed Scheduling for Asynchronous Message Passing. In *Proceedings of the Twenty-Second Conference on Uncertainty in Artificial Intelligence (UAI'06)*. AUAI Press, 165–173. https://doi.org/10.48550/ARXIV.1206.6837

[26] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. 2011. Dark Silicon and The End of Multicore Scaling. In *Proc. of the International Symposium on Computer Architecture (ISCA-38)*. ACM/IEEE, 122–134. https://doi.org/10.1109/MM.2012.17

[27] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R.M. Badia, E. Ayguade, J. Labarta, and M. Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. In *Proc. of the International Symposium on Microarchitecture (MICRO-43)*. IEEE/ACM, 89–100. https://doi.org/10.1109/MICRO.2010.13

[28] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM* 34, 3 (1987), 596–615. https://doi.org/10.1145/28869.28874

[29] Thomas Haigh, Mark Priestley, and Crispin Rope. 2014. Reconsidering the Stored-Program Concept. *IEEE Annals of the History of Computing* 36, 1 (2014), 4–17. https://doi.org/10.1109/MAHC.2013.56

[30] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, 58–69. https://doi.org/10.1145/384265.291020

[31] Peter Hart, Nils Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136

[32] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 166–177. https://doi.org/10.1145/2612669.2612697

[33] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 3–12. https://doi.org/10.1145/2038037.1941557

[34] Muhammad Amber Hassaan, Donald Nguyen, and Keshav Pingali. 2015. Kinetic Dependence Graphs. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*. ACM, 457–471. https://doi.org/10.1145/2775054.2694363

[35] David R. Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 3 (1985), 404–425. https://doi.org/10.1145/3916.3988

[36] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. 2016. Data-centric execution of speculative parallel programs. In *Proc. of the International Symposium on Microarchitecture (MICRO-49)*. IEEE/ACM, 1–13. https://doi.org/10.1109/MICRO.2016.7783708

[37] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proc. of the International Symposium on Microarchitecture (MICRO-48)*. IEEE/ACM, 228–241. https://doi.org/10.1145/2830772.2830777

[38] 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), 105–117. https://doi.org/10.1109/MM.2016.12

[39] Mark C. Jeffrey, Victor A. Ying, Suvinay Subramanian, Hyun Ryong Lee, Joel Emer, and Daniel Sanchez. 2018. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *Proc. of the International Symposium on Microarchitecture (MICRO-51)*. IEEE/ACM, 217–230. https://doi.org/10.1109/MICRO.2018.00026

[40] David S. Johnson. 1974. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.* 9, 3 (1974), 256–278. https://doi.org/10.1016/S0022-0000(74)80044-9

[41] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. ACM, 211–222. https://doi.org/10.1145/605397.605420

[42] Venkata Krishnan and Josep Torrellas. 1999. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. Comput.* 48, 9 (1999), 866–880. https://doi.org/10.1109/12.795218

[43] Joseph B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.* 7 (1956), 48–50. https://doi.org/10.2307/2033241

[44] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proc. of the Conference on Programming Language Design and Implementation*

(PLDI). ACM.

[45] Charles Leiserson and Tao Schardl. 2010. A work-efficient parallel breadth-first search algorithm. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 303–314. https://doi.org/10.1145/1810479.1810534

[46] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority queues are not good concurrent priority schedulers. In *Proc. of the European Conference on Parallel Processing (Euro-Par)*. Springer Berlin Heidelberg, 209–221. https://doi.org/10.1007/978-3-662-48096-0_17

[47] 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. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200. https://doi.org/10.1145/1064978.1065034

[48] Michael W. Mahoney, Anirban Dasgupta, Jure Leskovec, and Kevin J. Lan. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009). https://doi.org/10.1080/15427951.2009.10129177

[49] Fragkiskos D. Malliaros, Christos Giatsidis, Apostolos N. Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: theory, algorithms and applications. *The VLDB Journal* 29 (2020), 61–92. https://doi.org/10.1007/s00778-019-00587-4

[50] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427. https://doi.org/10.1145/2402.322385

[51] U. Meyer and P. Sanders. 2003. Delta-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[52] Lucas Morais, Vitor Silva, Alfredo Goldman, Carlos Alvarez, Jaume Bosch, Michael Frank, and Guido Araujo. 2019. Adding Tightly-Integrated Task Scheduling Acceleration to a RISC-V Multi-core Processor. In *Proc. of the International Symposium on Microarchitecture (MICRO-52)*. IEEE/ACM, 861–872. https://doi.org/10.1145/3352460.3358271

[53] Flaviano Morone, Kate Burleson-Lesser, H. A. Vinutha, Srikanth Sastry, and Hernán A. Makse. 2019. The jamming transition is a k-core percolation transition. *Physica A: Statistical Mechanics and its Applications* 516 (2019), 172–177. https://doi.org/10.1016/j.physa.2018.10.035

[54] Flaviano Morone, Gino Del Ferraro, and Hernán A. Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nature Physics* 15 (2019), 95–102. https://doi.org/10.1038/s41567-018-0304-8

[55] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI-11)*. USENIX, 511–524.

[56] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proc. of the Symposium on Operating Systems Principles (SOSP-24)*. ACM, 456–471. https://doi.org/10.1145/2517349.2522739

[57] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News* 33, 5 (2005), 45–50. https://doi.org/10.1145/1127577.1127587

[58] 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. of the Conference on Programming Language Design and Implementation (PLDI)*. ACM, 12–25. https://doi.org/10.1145/1993498.1993501

[59] D. N. Pnevmatikatos and G. S. Sohi. 1994. Guarded Execution and Branch Prediction in Dynamic ILP Processors. In *Proc. of the International Symposium on Computer Architecture (ISCA-21)*. ACM/IEEE, 120–129. https://doi.org/10.1145/191995.192022

[60] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-Queues Can Be State-of-the-Art Priority Schedulers. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 353–367. https://doi.org/10.1145/3503221.3508432

[61] R. C. Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

[62] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti Sarangi, James Tuck, and Josep Torrellas. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proc. of the International Conference on Supercomputing (ICS'05)*. ACM, 219–228. https://doi.org/10.1145/1088149.1088178

[63] 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. of the International Conference on Supercomputing (ICS'05)*. ACM, 179–188. https://doi.org/10.1145/1088149.1088173

[64] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 80–82. https://doi.org/10.1145/2755573.2755616

[65] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287. https://doi.org/10.1016/0378-8733(83)90028-X

[66] Mohsin Shan and Omer Khan. 2021. Accelerating Concurrent Priority Scheduling Using Adaptive in-Hardware Task Distribution in Multicores. *IEEE Computer Architecture Letters (CAL)* 20, 1 (2021), 17–21. https://doi.org/10.1109/LCA.2020.3045670

[67] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis — Patterns, Anomalies and Algorithms. In *Proc of the International Conference on Data Mining (ICDM)*. IEEE, 469–478. https://doi.org/10.1109/ICDM.2016.0058

[68] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention through Priority Updates. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 152–163. https://doi.org/10.1145/2486159.2486189

[69] 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. of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 68–70. https://doi.org/10.1145/2312005.2312018

[70] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar processors. In *Proc. of the International Symposium on Computer Architecture (ISCA-22)*. ACM/IEEE, 414–425. https://doi.org/10.1145/223982.224451

[71] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A scalable approach to thread-level speculation. In *Proc. of the International Symposium on Computer Architecture (ISCA-27)*. ACM/IEEE, 1–12. https://doi.org/10.1145/339647.339650

[72] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An execution model for fine-grain nested speculative parallelism. In *Proc. of the International Symposium on Computer Architecture (ISCA-44)*. ACM/IEEE, 587–599. https://doi.org/10.1145/3079856.3080218

[73] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé, and Mateo Valero. 2017. General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 244–253. https://doi.org/10.1109/IPDPS.2017.48

[74] Mikkel Thorup. 2000. On RAM Priority Queues. *SIAM J. Comput.* 30, 1 (2000), 86–109. https://doi.org/10.1137/S0097539795288246

[75] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. 1999. The Superthreaded Processor Architecture. *IEEE Trans. Comput.* 48, 9 (1999), 881–902. https://doi.org/10.1109/12.795219

[76] Jean Vuillemin. 1978. A Data Structure for Manipulating Priority Queues. *Commun. ACM* 21, 4 (1978), 309–315. https://doi.org/10.1145/359460.359478

[77] 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), 15–31. https://doi.org/10.1109/MM.2007.4378780

[78] Maurice V. Wilkes and William Renwick. 1949. The EDSAC - an Electronic Calculating Machine. *Journal of Scientific Instruments* 26, 12 (1949), 385–391. https://doi.org/10.1088/0950-7671/26/12/301

[79] J. W. J Williams. 1964. Algorithm 232 Heapsort. *Commun. ACM* 7, 6 (1964), 347–349. https://doi.org/10.1145/512274.512284

[80] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. 2015. The Lock-Free k-LSM Relaxed Priority Queue. In *Proc. of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 277–278. https://doi.org/10.1145/2688500.2688547

[81] Stefan Wuchty and Eivind Almaas. 2005. Peeling the yeast protein network. *Proteomics* 5 (2005), 444–449. Issue 2. https://doi.org/10.1002/pmic.200400962

[82] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *Proc. International Conference on Data Mining (ICDM)*. IEEE, 745–754. https://doi.org/10.1109/ICDM.2012.138

[83] Fahimeh Yazdanpanah, Carlos Álvarez, Daniel Jiménez-González, Rosa M. Badia, and Mateo Valero. 2015. Picos: A hardware runtime architecture support for OmpSs. *Future Generation Computer Systems* 53 (December 2015), 130–139. https://doi.org/10.1016/j.future.2014.12.010

[84] 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. of the International Symposium on High Performance Computer Architecture (HPCA-13)*. IEEE, 261–272. https://doi.org/10.1109/HPCA.2007.346204

[85] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2019. Understanding Priority-Based Scheduling of Graph Algorithms on a Shared-Memory Platform. In *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*. ACM, 1–14. https://doi.org/10.1145/3295500.3356160

[86] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling sequential code for effective speculative parallelization in hardware. In *Proc. of the International Symposium on Computer Architecture (ISCA-47)*. ACM/IEEE, 159–172. https://doi.org/10.1109/ISCA45697.2020.00024

[87] Guowei Zhang, Webb Horn, and Daniel Sanchez. 2015. Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems. In *Proc. of the International Symposium on Microarchitecture (MICRO-48)*. IEEE/ACM, 13–25. https://doi.org/10.1145/2830772.2830774

[88] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*. IEEE. https://doi.org/10.48550/ARXIV.1911.07260

[89] Tingzhe Zhou, Maged Michael, and Michael Spear. 2019. A Practical, Scalable, Relaxed Priority Queue. In *Proc. of the International Conference on Parallel Processing (ICPP)*. ACM, 1–10. https://doi.org/10.1145/3337821.3337911