Optimizing Throughput Architectures for Speculative Parallelism

by

Weeraratna Patabendige Maleen Hasanka Abeydeera

B.Sc. (Eng) in Electronic and Telecommunication Engineering
University of Moratuwa, Sri Lanka, 2014

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author .. Signature redacted ....
Department of Electrical Engineering and Computer Science
May 19, 2017

Certified by. Signature redacted ....
Daniel Sanchez
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ... Signature redacted ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Optimizing Throughput Architectures for Speculative Parallelism

by

Weeraratna Patabendige Maleen Hasanka Abeydeera

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2017, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Throughput-oriented architectures, like GPUs, use a large number of simple cores and rely on application-level parallelism, using multithreading to keep the cores busy. These architectures work well when parallelism is plentiful but work poorly when its not. Therefore, it is important to combine these techniques with other hardware support for parallelizing challenging applications.

Recent work has shown that speculative parallelism is plentiful for a large class of applications that have traditionally been hard to parallelize. However, adding hardware support for speculative parallelism to a throughput-oriented system leads to a severe pathology: aborted work consumes scarce resources and hurts the throughput of useful work.

This thesis develops a technique to optimize throughput-oriented architectures for speculative parallelism: tasks should be prioritized according to how speculative they are. This focuses resources on work that is more likely to commit, reducing aborts and using speculation resources more efficiently. We identify two on-chip resources where this prioritization is most likely to help, the core pipeline and the memory controller.

First, this thesis presents speculation-aware multithreading (SAM), a simple policy that modifies a multithreaded processor pipeline to prioritize instructions from less speculative tasks. Second, we modify the on-chip memory controller to prioritize requests issued by tasks that are earlier in the conflict resolution order.

We evaluate SAM on systems with up to 64 SMT cores. With SAM, 8-threaded in-order cores outperform single-threaded cores by 2.41× on average, while a speculation-oblivious policy yields a 1.91× speedup. SAM also reduces wasted work by 43%. Unlike at the core, we find little performance benefit from prioritizing requests at the memory controller. The reason is that speculative execution works as a very effective prefetching mechanism, and most requests, even those from tasks that are ultimately aborted, do end up being useful.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my research advisor Daniel Sanchez, who has been a constant source of encouragement for me during these past two years. Always friendly and approachable, Daniel's help was extremely useful in coming up with the research problem that led to this thesis, setting up the evaluation infrastructure, and finally writing down the results.

A special thanks goes my collaborators Mark Jeffrey, Suvinay Subramanian, and Professor Joel Emer for their help. A shout out also goes to the rest of the Sanchez group, Nathan, Harshad, Po-An, Anurag, Guowei, Victor, Ryan, Nosayba, and Yee-Ling for their feedback during various stages of this work.

I am forever indebted to my parents for all their sacrifices they made to mold me into who I am today. When times were tough they made sure that I did not feel it and somehow provided for my education. I also want to thank the love-of-my-life and soon-to-be wife, Sandamali, for standing beside me at all times.

The stress and loneliness of studying at a foreign university would have been too much if not for the weekly social gatherings with Charith, Savithru, Narmada, Thulith, Navodya, Tharanga, and Chamithri. They made sure that there was always something to look forward to when the going got tough.

Last but not least, I want to thank all the people of Sri Lanka, who by virtue of the free education system, paid for my school and undergraduate education and allowed me to be who I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Throughput-oriented processors (e.g., Xeon Phi [70], Niagara [40], GPUs [38]) in contrast to traditional scalar microprocessors, use massive amounts of simple cores, and rely on hardware multithreading to maximize the rate of completed work. These systems rely on application level parallelism to keep the cores busy. Throughput-oriented architectures are very efficient when parallelism is plentiful because they avoid the complexity and overhead of high-performance cores. However, when application parallelism is scarce, the system becomes underutilized. Therefore, it is important to combine throughput-oriented architectures with hardware techniques that allow extracting more parallelism from applications.

Hardware support for speculative parallelism would make these systems broadly useful. For applications whose tasks have dependences that are hard to analyze statically, prior techniques such as Thread-Level Speculation (TLS) and Hardware Transactional Memory (HTM) have proposed to run tasks speculatively. Hardware detects dependence violations among tasks dynamically and re-execute conflicting tasks in the correct order. More recently, Swarm [32,33] generalizes these techniques to extract substantially more parallelism from these applications; for instance, the route planning algorithm `astar`, which prior work has failed to parallelized statically, scales linearly with speculative execution up to 128 cores. Hence, running these applications in throughput processors has become appealing.

However, when hardware support for speculative execution is applied for throughput-oriented systems, tasks that are likely to abort consume resources and hurt the throughput of less-speculative ones. This effect causes two pathologies: the amount of mis-speculated work grows, and the time spent holding speculation resources increases, causing more stalls. Therefore, unlike in non-speculative systems, where throughput is measured in terms of total work, speculative systems should maximize the rate of committed work.

1.1 Contributions

The key insight of this thesis is that throughput architectures can be optimized for speculative parallelism by prioritizing tasks according to how speculative they are. For TLS and other schemes that support ordered parallelism [25, 33, 71, 74], where the program dictates the execution order of speculative tasks, this order directly determines how speculative each task is. For HTM and other schemes that support unordered parallelism [16,26,46], where any execution order is valid, it is less clear how speculative each task is. However, we observe that HTM conflict resolution policies often enforce an order among transactions on the fly. We can leverage this order to prioritize tasks.

There are multiple resources in a system that are shared among multiple concurrent tasks, including the core, caches, on-chip network, and memory controllers. A key question is where this prioritization would be most useful. We focus on the core pipeline, which is often saturated by multithreading, and the memory controller, which is often the most contended resource in the memory system.

1.1.1 Prioritization at the Core

We present *speculation-aware multithreading (SAM)*, a simple policy that modifies a multithreaded processor pipeline to prioritize instructions from less-speculative tasks (Sec. 2.2). SAM avoids pipeline interference from more- to less-speculative tasks, reducing wasted work. And because less-speculative tasks commit earlier, SAM also makes more effective use of speculation resources.

We design SAM variants for in-order and out-of-order cores. We find that SAM is much more effective than prior SMT policies that aim to maximize pipeline efficiency, like ICount [77]. We also present a simple adaptive policy that achieves SAM's low aborts when contention is high, and ICount's high pipeline efficiency when contention is low.

SAM improves the performance benefit of multithreaded cores on speculative parallel programs. We demonstrate SAM on an architecture that supports ordered and unordered speculative parallelism (Chapter 3). On a 64-core system with 2-wide issue in-order SMT cores, with SAM, 8-threaded cores outperform single-threaded cores by 2.41× on average, while speculation-oblivious round-robin achieves 1.91× improvement. SAM also reduces wasted work by 43%, making speculative execution more efficient. With out-of-order execution, 8-threaded cores improve performance over single-threaded cores by 1.53× with SAM vs only 1.16× with ICount, and SAM reduces wasted work by 81% (Chapter 5).

1.1.2 Prioritization at the Memory Controller

We also investigate whether prioritizing requests at the memory controller according to their conflict resolution priorities can provide performance benefits. Like SAM, deprioritizing requests from more speculative tasks inhibits their progress and less work is wasted if they were to ultimately abort. Unlike SAM, where each core makes a local decision on which thread to issue from, the memory controller is a global resource and can select among all threads in the system.

However, we find that speculative execution works as a very effective prefetching mechanism. Speculatively running tasks far ahead in program order helps to bring the data they access into on-chip caches. When a task is aborted, the data it fetched is usually useful for other tasks that commit. Hence, from the point of view of the memory controller, requests served for aborted tasks are rarely wasted work.

We corroborate this observation with simulation, where we see no significant performance advantage by prioritizing requests at the memory controller. Therefore we conclude that memory system prioritization would yield negligible benefits.

Chapter 2

Speculation-Aware Multithreading

2.1 Motivation

This section explores the interplay between multithreaded cores and speculative tasks, by analyzing the behavior of a few representative applications as the number of threads per core increases. This analysis identifies the consequences of speculation-oblivious multithreading, and motivates the need for a speculation-aware multithreading policy.

For these experiments, we use an architecture that performs well on both ordered and unordered speculative workloads. This architecture is based on Swarm [33, 34], a recent proposal for ordered parallelism that uses large hardware task queues to speculate far ahead of the earliest active task. Although Swarm was primarily focused on ordered speculative workloads, our baseline extends it to more effectively support unordered speculative workloads. This is accomplished by adopting the conflict resolution policy proposed by Wait-N-GoTM [30], which, upon a conflict, adaptively decides whether to forward speculative data or to stall the requester, and orders tasks lazily. This policy reduces aborts under contention, especially for unordered benchmarks. Chapter 3 describes this baseline architecture in detail, but in-depth knowledge is not required to understand the following analysis.

The baseline uses 2-wide issue, in-order cores similar to those of Cavium ThunderX [24]. Cores use SMT: at each cycle, the core can issue up to two micro-ops from one or two threads. When multiple threads have issuable micro-ops, a speculation-oblivious *round-robin* policy selects among them.

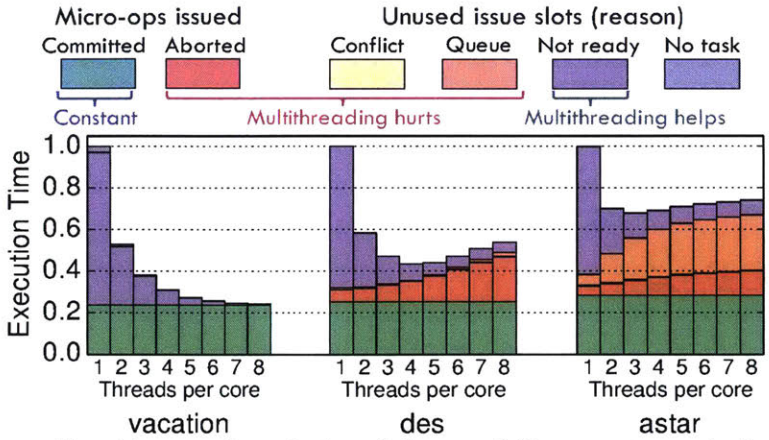


Figure 2-1: Execution time and cycle breakdown of three representative apps running on 64-core systems with 1 to 8 threads per core (lower is better).

Fig. 2-1 shows how the number of threads per core affects performance on a 64-core system (Chapter 4 details our methodology). Each 8-bar group reports results for a single application, using from 1 to 8 threads per core. We consider an unordered application, `vacation`, and two ordered applications, `des` and `astar`. The height of each bar is execution time relative to that of single-threaded cores (lower bars are better). Moreover, each bar shows the breakdown of how cores spend cycles:

- Cycles where micro-ops are issued by tasks that:
  - perform useful work that will be *committed*, or
  - are performing work that will later be *aborted*.
- Cycles where no micro-op is issued, because:
  - data or structural dependences among a thread's instructions result in all micro-ops being *not ready*,
  - an inter-task data-dependence *conflict* has stalled a thread's task,
  - a thread is stalled because a speculation resource is full, such as the task or commit *queue*, or
  - a thread has no instructions because it has *no task* to run.

Among these categories, multithreading is aimed at reducing *not ready* in order to increase the number of cycles where micro-ops are issued. This is beneficial when the effect is an increased rate of *committed* micro-ops. However, we will show multithreading can also have the undesirable consequence of increasing cycles spent in *aborted*, *conflict*, and *queue*.

Multithreading can be highly beneficial: `vacation` in Fig. 2-1 shows that multithreading can dramatically increase performance. `vacation` has plentiful parallelism, but accesses main memory frequently. As a result, its time is spent either issuing instructions from tasks that later *commit*, or waiting (*not ready*) on data dependences caused by long-latency loads. With a single thread per core, the latter stalls waste 76% of issue slots. These stalls greatly decrease with multithreading. They are still significant with four threads per core, but become negligible at eight threads per core. With eight threads per core, `vacation` is 4.1× faster than on a single thread.

This result shows that multithreading can improve performance on speculative programs: these programs often have much more parallelism than the system has cores, and multithreading is a cheap way to put that parallelism to good use. Supporting eight threads increases core area by about 30% [18], but quadruples performance in `vacation`. Although more threads yield diminishing returns, we find that the most resource-efficient configuration is often highly threaded.

However, speculation introduces two deleterious pathologies that can limit the benefits of multithreading:

Pathology 1: Increased aborts: `des` in Fig. 2-1 shows that multithreading can increase wasted work to the point of hurting performance. Like `vacation`, `des` with a single thread per core loses many issue slots to dependences among instructions. Unlike `vacation`, `des` has limited parallelism: with a single thread per core, 7% of issue slots are wasted on tasks that are later aborted. Aborts grow with the number of threads per core: with eight threads per core, 40% of issue slots are lost to aborted work. As a result, multithreading *hurts performance* beyond four threads per core.

It is well known that, when speculative applications have limited parallelism, increasing concurrency adds aborts and may hurt performance. However, prior work has shown this effect when increasing the number of cores [84], not the number of threads per core. This implies two critical differences. First, with multithreading, wasted work hurts performance much more quickly than when increasing the number of cores, because *tasks that will abort take execution resources away from tasks that will commit, slowing them down.* Second, with multithreading, there is a simple way to affect how instructions from different tasks share core resources: the issue policy. A speculation-aware issue policy can prioritize instructions from likely-to-commit tasks, improving their performance.

Pathology 2: Inefficient use of speculation resources: astar in Fig. 2-1 shows that multithreading can degrade performance by overloading speculation resources. Like the two previous applications, single-threaded astar loses over half of issue slots to instruction dependences, which multithreading could address. However, astar is an ordered application that stresses our baseline's commit queues. Commit queues hold the speculative state of tasks that finish execution but cannot yet commit, so that the core can run another task. When these commit queues fill up, however, cores cannot run more tasks, and stall. Fig. 2-1 shows that these queue stalls increase with the number of threads per core, and make multithreading degrade performance beyond three threads.

In general, adding threads increases pressure on speculation resources due to two compounding effects. First, more tasks are active, demanding more speculation resources. Second, multithreading increases the latency of individual tasks, so tasks hold speculative resources for longer. This is not limited to commit queues, e.g., BlueGene/Q runs out of transaction IDs more frequently with multiple threads per core [81].

In summary, wasted work and inefficient use of speculation resources have a substantial impact on the performance of multithreading. These observations lead to speculation-aware multithreading (SAM). SAM prioritizes the execution of tasks with a higher conflict resolution priority. SAM reduces wasted work because it focuses execution resources on tasks that are more likely to commit. And SAM also reduces the time speculation resources are held, because tasks with a higher conflict resolution priority commit earlier. Though simple, SAM is highly effective at addressing these pathologies.

2.2 Speculation-Aware Multithreading

The speculation-aware multithreading (SAM) policy prioritizes each thread according to the conflict resolution priority of the speculative task that the thread is currently running.

We describe SAM's mechanisms for a generic conflict resolution policy (we discuss our baseline's policy in Chapter 3). A conflict resolution policy establishes an implicit or explicit priority order among speculative tasks, and resolves conflicts among tasks following this priority. For example, under most policies, lower-priority tasks cannot abort higher-priority tasks.

There is a wide variety of conflict resolution policies [10, 30, 46, 67], both in terms of the information used to prioritize tasks (age, work done so far, etc.) and the corrective actions taken upon a conflict (stalling or aborting a task, or forwarding data). In general, two characteristics are relevant for SAM. First, a task's conflict resolution priority can change while the task runs (e.g., upon a conflict with another task). Therefore, SAM interfaces with the conflict resolution policy to receive these frequent priority updates and immediately adjust thread priorities. Second, two tasks may have the same priority (e.g., if they are unordered and have not encountered any conflicts). Therefore, SAM breaks ties among same-priority threads using a secondary policy, such as round-robin or ICount.

We describe SAM's implementation for in-order and out-of-order cores, and present experiments that show that *(i)* the key reason SAM works well is because it devotes resources to tasks that are more likely to commit, and *(ii)* SAM works better the more aggressively it can prioritize a single thread.

2.2.1 SAM on in-order cores

Fig. 2-2 shows the in-order core we use and the changes needed to support SAM. Our implementation performs issue-stage prioritization. Each cycle, the issue stage selects among ready micro-ops from all threads. Priorities are absolute: ready micro-ops from a higher-priority thread are always selected over those of lower-priority threads. Ready micro-ops from same-priority threads share slots using a round-robin policy.

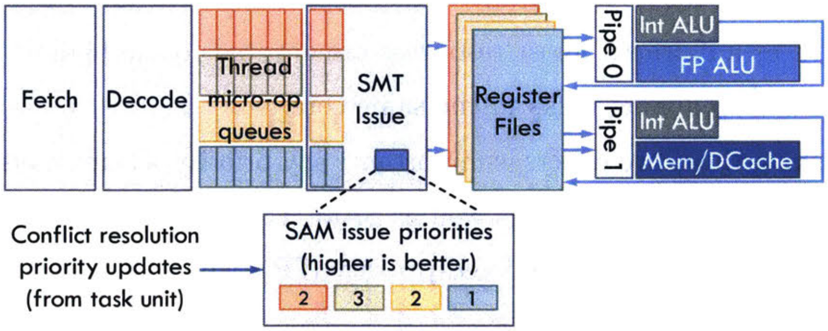


Figure 2-2: In-order core with SAM modifications.

This prioritized issue scheme is simple and available in commercial systems [11,23]. The key problem that SAM addresses is how to set thread priorities to maximize the benefits of multithreaded cores on speculative systems.

SAM vs other policies: Fig. 2-3 compares the performance of different multithreading policies on a system with 64 in-order cores and 1, 2, 4, or 8 threads per core. Each bar's height denotes normalized execution time over the single-threaded core, averaged across all 16 applications in our suite (lower is better). Each bar also shows the breakdown of issue slots, following the same nomenclature as Sec. 2.1.

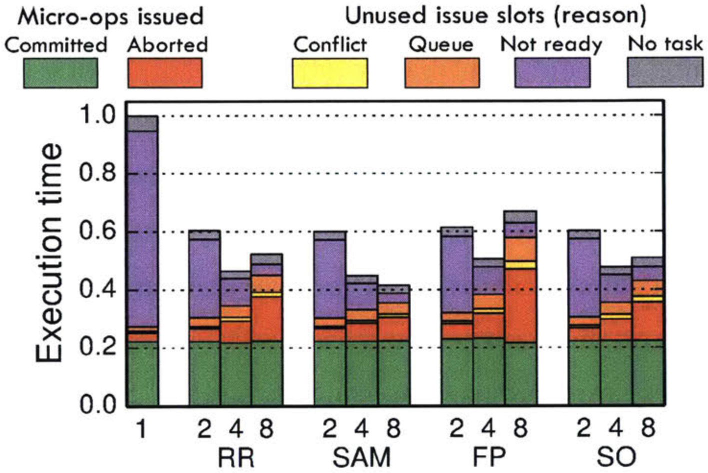


Figure 2-3: Performance and execution time breakdown of different issue policies across all applications, on a system with 64 in-order cores and 1/2/4/8 threads per core (lower is better).

Fig. 2-3 shows that SAM significantly outperforms the baseline *round-robin* policy (RR).[1] SAM reduces aborts, queue stalls, and conflicts. As we will see in Chapter 5, these benefits are consistent across all applications.

Two effects could explain SAM's improvement over RR. First, SAM prioritizes tasks that are more likely to commit. Second, SAM, and in fact any prioritization policy, introduces unfairness: most resources are devoted to the highest-priority task, reducing the overlap among tasks in the same core.

Distinguishing these two effects is important: any priority scheme causes unfairness, so simpler policies could perform as well as SAM. To this end, Fig. 2-3 also includes two simple prioritization policies: fixed-priority (FP), where each thread in the core

[1] We have evaluated speculation-oblivious policies beyond RR, like ICount, but they make nearly no difference on an in-order core, as we will see next.

uses a fixed priority that is preserved across tasks; and start-order (SO), which gives higher priority to older tasks.

FP performs worst, showing that prioritizing differently than the conflict resolution priority is a poor strategy: FP often gives resources to tasks that are likely to abort, wasting much more work than any other policy. At 8 threads per core, FP is 29% slower than RR. SO is typically better than RR but worse than SAM, outperforming RR by just 2% on average. SO performs better than FP because start order is often similar to conflict resolution order. These experiments show that prioritizing likely-to-commit work is the dominant effect.

In summary, simpler order policies perform worse than directly enforcing conflict resolution priorities. One may wonder whether a more sophisticated policy would perform better, e.g., using prediction to better estimate how likely a task is to commit. However, *if such a predictor exists, we argue that it should be used to alter the conflict resolution priority directly.*

Fairness and forward progress: Finally, note that, while priorities may cause long-term unfairness and even prevent forward progress in non-speculative systems [13], SAM does not suffer from these problems because conflict resolution policies always guarantee that every task can eventually become the system's highest-priority task [5, 10, 46].

2.2.2 SAM on out-of-order cores

SAM is unfair by design—it prioritizes one or a few threads, rather than sharing resources equitably among threads. On in-order cores, thread priorities have little effect on pipeline efficiency. But priorities can affect the throughput of out-of-order (OoO) SMT cores, for two reasons:

- *Increased stalls:* Threads in an OoO core share limited issue buffer and reorder buffer (ROB) entries, as well as physical (renamed) registers. These resources are acquired by micro-ops before they are ready to issue. Therefore, prioritizing one thread may clog these resources with dependent micro-ops that will take a long time to become ready, causing stalls. Prior OoO SMT issue policies like ICount [77] address this issue by prioritizing threads that use these resources better. This is not a problem on in-order cores because prioritization is only done among *ready* micro-ops.

- *Increased wrong-path execution:* OoO cores can execute micro-ops far past a mispredicted branch. These wrong-path micro-ops waste execution resources. In SMT cores, if resources are shared fairly among threads, wrong-path execution becomes less frequent, because each thread has fewer micro-ops in flight (and thus does not execute as far past unresolved branches). But this reduction does not materialize if we prioritize a particular thread rather than sharing resources fairly. This is not a problem on in-order cores because a non-issuable branch prevents subsequent instructions from being issued (e.g., our in-order core resolves branches at issue, so it avoids wrong-path issues, though it does perform wrong-path fetches and decodes).

Despite these handicaps, we find that *prioritizing instructions from likely-to-commit tasks is the first-order constraint for OoO cores.* Therefore, our SAM implementation performs aggressive prioritization.

Basic SAM policy: Fig. 2-4 shows our OoO core SAM implementation. Each cycle, if there are free issue buffer, ROB, and renamed register entries, the issue stage injects up to two decoded micro-ops into the unified issue buffer. SAM performs prioritization at this point, always selecting micro-ops from higher-priority threads. SAM breaks ties among same-priority threads using ICount (i.e., it selects micro-ops from the thread with the fewest micro-ops in flight). This way, SAM retains ICount's pipeline efficiency when tasks are undifferentiated.

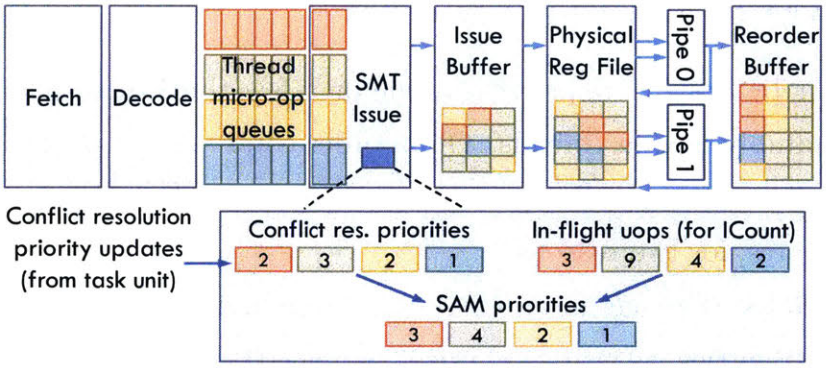


Figure 2-4: Out-of-order core with SAM modifications.

Unfairness is good: As shown in Fig. 2-4, in our specific design, all backend structures (issue buffer, ROB, physical registers, and load-store queues) are dynamically shared among threads rather than statically partitioned. The reason is that shared structures let SAM prioritize threads more aggressively.

Fig. 2-5 shows why this is a good idea by comparing the performance of statically-partitioned and dynamically-shared ROBs under ICount and SAM. We simulate 2-wide issue cores with 36 issue buffer and 72 ROB entries (see Chapter 4 for details). Each bar shows the normalized execution time over the single-threaded core, averaged across all benchmarks.

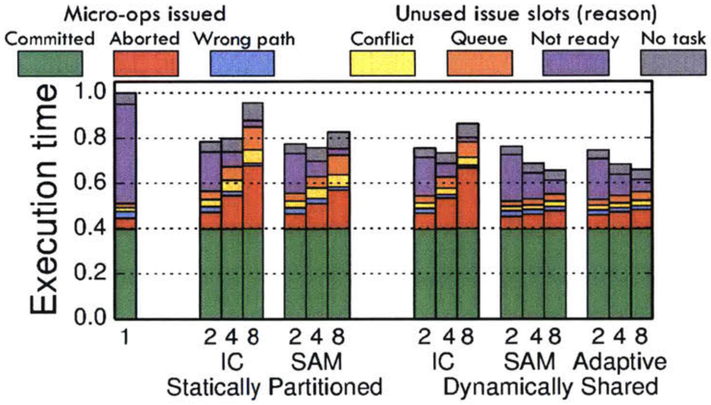


Figure 2-5: Performance of ICount, basic SAM, and adaptive SAM with statically-partitioned vs dynamically-shared ROBs.

Fig. 2-5 shows that, with more threads, ICount suffers from more aborts and queue and conflict stalls. These hurt performance with more threads, despite ICount's increased pipeline utilization (fewer cycles lost to wrong-path or not-ready micro-ops). Partitioned and shared ROBs show the same trend.

SAM ameliorates these pathologies, but the type of ROB impacts its effectiveness. With partitioned ROBs, as threads grow SAM still suffers from increased aborts and queue/conflict stalls, although at a lower rate than ICount. This happens because the highest-priority thread fills its ROB partition and lets micro-ops from other, more speculative threads be issued.

With a shared ROB, however, SAM can fill the issue buffer with micro-ops from a single thread. As a result, SAM keeps cycles lost to aborts and queue/conflict stalls nearly flat. This comes at the price of higher wrong-path micro-ops and not-ready stalls. But these inefficiencies are secondary, and SAM is thus most effective when it can prioritize most aggressively.

Finally, note that SAM's desire for prioritization makes our core deviate from typical designs, which seek some amount of fairness among threads. For example, dynamically shared ROBs are relatively rare (e.g., the EV8 used a shared ROB [20], but modern Intel cores use partitioned ROBs). And our results contradict prior work by Raasch and Reinhardt [59], who find that partitioned vs shared ROBs make little difference, because they implicitly focused on fair policies.

Adaptive SAM policy: The above results show that, *on average*, it is better to prioritize aggressively. However, applications with rare aborts and little contention can still benefit from ICount's higher pipeline efficiency. To this end, we implement a simple policy that combines the benefits of SAM and ICount. This policy keeps running counts of cycles lost to task-level speculation $(aborted + conflict + queue)$ and pipeline inefficiencies. $(not\ ready - wrong\ path)$. If cycles lost to task-level speculation dominate, the core uses SAM; if cycles lost to pipeline inefficiencies dominate, the core uses ICount. Fig. 2-5 shows that this adaptive policy slightly improves on the basic SAM policy at 2 and 4 threads.

Chapter 3

Baseline Speculative Architecture

We implement the SAM policy on a baseline speculative architecture that performs well on both ordered and unordered programs. This lets us evaluate our techniques with a broader range of speculative programs than if we used a TLS or HTM baseline. To support ordered and unordered programs, this architecture is based on Swarm [33,34]. To reduce aborts under contention and make the system more efficient on unordered benchmarks, we adopt the conflict resolution techniques from Wait-N-GoTM [30]. Although we evaluate SAM within this baseline, SAM solves a general problem and should benefit any other HTM and TLS schemes that use multithreaded cores.

Sec. 3.1 and Sec. 3.2 present Swarm's main features (see prior work [33,34] for details). Sec. 3.3 describes the Swarm + Wait-N-GoTM conflict resolution policy. Sec. 3.4 extends Swarm's conflict detection mechanisms to cheaply support multithreaded cores, in a way similar to BulkSMT [57].

3.1 Swarm Execution Model

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

Swarm exposes its execution model through a simple API. Listing 3.1 illustrates this API by showing the Swarm implementation of `des`, a discrete event simulator for digital circuits adapted from Galois [28, 55].

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

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

Listing 3.1: Swarm implementation of discrete event simulation for digital circuits.

Each task runs a function that takes a timestamp and an arbitrary number of additional arguments. Listing 3.1 defines one task function, **desTask**, which simulates a signal toggling at a gate input. Tasks can create child tasks by calling **swarm::enqueue** with the appropriate task function, timestamp, and arguments. In our example, if an input toggle causes the gate output to toggle, **desTask** enqueues child tasks for all the gates connected to this output. Finally, a program invokes Swarm by enqueuing some initial tasks with **swarm::enqueue** and calling **swarm::run**, which returns control when all tasks finish. For example, Listing 3.1 enqueues a task for each input waveform, then starts the simulation.

Swarm's execution model supports both TLS-style ordered speculation by choosing timestamps that reflect the serial order as in prior work [63], and TM-style unordered speculation by using an equal timestamp for all tasks. Moreover, Swarm's execution model generalizes TLS by *decoupling task creation and execution orders*: whereas in prior TLS schemes a task could only spawn speculative tasks that are immediate successors [25, 26, 63, 71, 73], a Swarm task can create child tasks with any timestamp equal or higher than its own. This allows programs to convey new work to hardware as soon as it is discovered instead of in the order it needs to run, exposing a large amount of parallelism for ordered irregular applications [33].

3.2 Swarm Microarchitecture

Swarm uncovers parallelism by executing tasks speculatively and out of order. To uncover enough parallelism, Swarm can speculate thousands of tasks ahead of the earliest active (unfinished) task. Swarm introduces modest changes to a tiled, cache-coherent multicore, shown in Fig. 3-1. Each tile has a group of multithreaded cores, each with its own private L1 cache. All cores in a tile share an L2 cache, and each tile has a slice of a fully-shared L3 cache. Every tile is augmented with a *task unit* that queues, dispatches, and commits tasks.

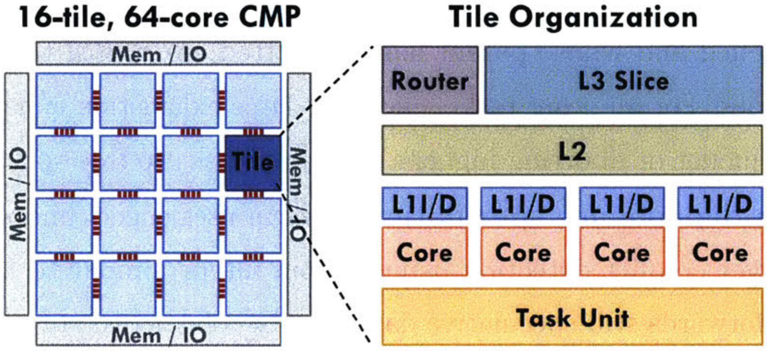


Figure 3-1: Swarm CMP and tile configuration.

Swarm hardware efficiently supports fine-grain tasks and a large speculation window through four main mechanisms: low-overhead hardware task management, large task queues, scalable data-dependence speculation mechanisms, and high-throughput ordered commits.

Hardware task management: Each tile's task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Swarm executes every task speculatively, except the earliest active task. To uncover enough parallelism, task units can dispatch any available task to cores, no matter how distant in timestamp order. A task can run even if its parent is still speculative.

Each task is represented by a task descriptor that contains its function pointer, timestamp, and arguments. Threads dequeue tasks for execution in timestamp order from the local task unit. Successful dequeues initiate speculative execution at the task's function pointer and make the task's timestamp and arguments available in registers. A thread may stall if there is no task to dequeue. Tasks create child tasks and enqueue them to a tile.

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

Together, these queues implement a task-level reorder buffer.

Task and commit queues support tens of speculative tasks per core (e.g., 128 task queue entries and 32 commit queue entries per core) to implement a large window of speculation (e.g., 8192 tasks in the 64-core CMP in Fig. 3-1). Nevertheless, because programs can enqueue tasks with arbitrary timestamps, task and commit queues can fill up. This requires some simple actions to ensure correct behavior. Tasks that have not been dequeued and whose parent has committed are *spilled* to memory to free task queue entries. For all other tasks, queue resource exhaustion is handled by either stalling the enqueuer or aborting higher-timestamp tasks to free space [33].

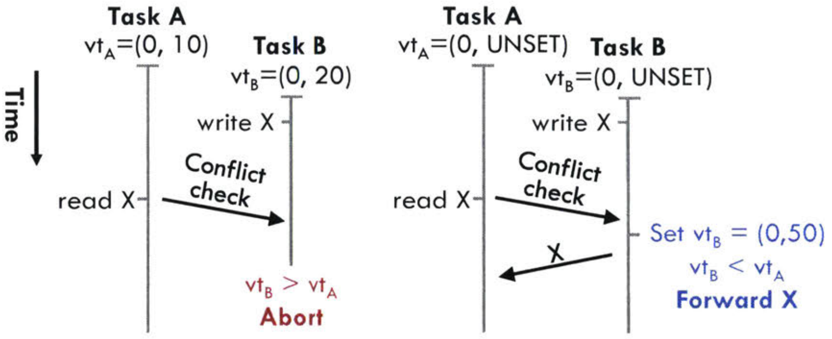
Scalable data-dependence speculation: Swarm uses eager (undo-log-based) versioning and eager conflict detection using Bloom filters, similar to LogTM-SE [83]. Swarm always forwards still-speculative data read by a later task; on an abort, Swarm aborts only descendants and data-dependent tasks.

High-throughput ordered commits: Finally, Swarm adapts the virtual time algorithm [31] to achieve high-throughput ordered commits. Tiles periodically communicate with an arbiter (e.g., every 200 cycles) to discover the earliest unfinished task in the system. All tasks that precede this earliest unfinished task can safely commit. This scheme achieves high commit rates, up to multiple tasks per cycle on average, which allows fine-grain ordered tasks, as short as a few tens of cycles.

3.3 Conflict Resolution Policy

Our key hypothesis is that thread issue priority and conflict resolution ordering should be coordinated. Therefore, it is important that we use a conflict resolution policy that does not overly restrict task ordering. Unfortunately, Swarm as proposed in prior work overly restricts conflict resolution order among unordered tasks. Furthermore, since multithreading often increases wasted work (Sec. 2.1), we should use a policy that minimizes aborts. Swarm also violates this principle and causes more aborts than needed by always forwarding speculative data. We solve both these problems by adapting the key techniques from Wait-n-GoTM [30].

Lazy virtual time tiebreakers: Swarm's conflict resolution policy encodes task order using *virtual time*: the concatenation of a task's programmer-assigned time-



(a) Needless abort caused by eager virtual time tiebreakers

(b) Lazy tiebreakers order on first conflict, avoiding abort

Figure 3-2: Eager virtual time tiebreakers (used in original Swarm) vs lazy tiebreakers (used here).

stamp and a *tiebreaker*. Tiebreakers are unique and monotonically increasing, which guarantees forward progress and preserves parent-before-child order. A task's virtual time determines both its commit order and its conflict resolution order: on an access, the task aborts all conflicting higher-virtual time tasks; conversely, the task can be aborted by any lower-virtual time tasks.

The original Swarm protocol greedily assigns each task a unique tiebreaker when the task begins execution. When tasks have equal programmer-assigned timestamp, greedy tiebreaking restricts order and causes needless aborts. Fig. 3-2(a) shows such a needless abort: tasks A and B both have timestamp 0, and are assigned tiebreakers 10 and 20 when they start execution. Task B writes to address X first, then task A issues a read request. Because B is ordered after A, B must abort.

Drawing from Wait-n-GoTM [30], we instead assign tiebreakers lazily. Tasks start running without a tiebreaker, and are assigned one when they acquire a dependence with an equal-timestamp task. Fig. 3-2(b) shows how this works in our example: tasks A and B have no tiebreaker until task A requests X. At that point, task B, which already wrote X, acquires a tiebreaker and forwards X's data to A. Tasks without a tiebreaker always compare higher than equal-timestamp tasks with a tiebreaker. To preserve parent-before-child order, a parent acquires a tiebreaker when it creates its first equal-timestamp child. To preserve commit order, if a task finishes execution without a tiebreaker, it is assigned one. To guarantee forward progress, a task retains its tiebreaker until it commits.

Wait-n-GoTM employs a more sophisticated scheme, TimeTraveler [79], which uses lower and upper bounds that are progressively restricted upon conflicts. One can construct situations where TimeTraveler would avoid aborts that a single tiebreaker cannot. However, these situations are rare (e.g., they involve three or more tasks conflicting on different addresses), and we observe the benefit would be marginal: across all applications, 81% of accesses come from tasks without tiebreakers. Therefore, we opt for this simpler scheme.

Adaptively stalling vs forwarding: Suppose an access from task A conflicts with task B (e.g., A issues a read to a line that B previously wrote). If B has higher virtual time than A, B must be aborted. However, if B has a lower virtual time than A, there are two options: the system could forward B's speculatively-written data to A, or it could stall A until B finishes executing or commits. Forwarding can improve performance, but makes A dependent on B, causing it to abort on a *cyclic* dependence, i.e., if B writes the line again.

Most systems adopt a fixed policy: LogTM [46,83] and most early HTMs [17,26,61] always stall, while Swarm, DATM [62], and most other conflict-serializable HTMs [5, 57,58] always forward. Wait-n-GoTM improves on these designs by detecting what conflicts are likely to cause cyclic dependences and stalling only on those. We adopt Wait-n-GoTM's line-based predictor and training scheme, including one predictor per tile. This predictor is checked before the tile responds to a conflicting request. If the line is predicted to cause a cyclic dependence, the tile NACKs the request, stalling requester task A, and records the dependence in staller task B's log. When B finishes, the tile ACKs task A, which resumes execution when all stalls have been cleared (multiple tasks may stall a given request). This implements the Wait-N-GoTM-*wait* variant [30].

SAM prioritization: In this system, the task's virtual time is its conflict resolution priority. Therefore, SAM prioritizes each thread using its task's virtual time. Tasks with a lower virtual time are given higher priority, and tasks with equal virtual time are given equal priority. The core recomputes thread priorities when a thread dequeues a new task and when a task's virtual time is assigned a tiebreaker.

3.4 Multithreaded Cores

Finally, we modify the Swarm L1 caches to support multiple threads. We strive for simplicity, since L1 accesses must be fast. Each line has a single *safe bit* per thread context. Safe bits let multiple threads share the L1 without violating conflict check rules. An L1 hit can only be served from the L1 if the thread's safe bit is set. If unset, the core issues an L2 access, which causes a conflict check. When the request finishes, the safe bit is set. If the request is a write, the line's safe bits of all other threads are cleared. When a thread dequeues a new task, if its virtual time precedes the previous task's, the thread's safe bits for all L1 lines are flash-cleared (safe bits are kept otherwise, because conflict checks performed for a given virtual time are also valid for higher ones [33]).

Safe bits are similar to the access bits in BulkSMT-ORDER [57]. Unlike BulkSMT, which can detect conflicts and order tasks within the core, we defer all conflict detection to the tile for simplicity. Because tiles are small, tile-level checks are fast.

Chapter 4

Experimental Methodology

Modeled system: We use a cycle-accurate, event-driven simulator based on Pin [44, 53]. We use detailed core, cache, network, and main memory models, and simulate all speculative execution overheads (e.g., running mispeculating tasks until they abort, simulating conflict check and rollback delays and traffic, etc.). We model systems of up to 64 cores (Fig. 3-1) and 8 threads per core, with parameters given in Table 4.1.

We use 2-wide issue in-order and out-of-order cores, shown in Figs. 2-2 and 2-4. Cores run the x86-64 ISA. We use the instruction decoder and functional-unit latencies of zsim's core model, which have been validated against Nehalem [66]. Our in-order core is similar to Cavium ThunderX [24], while out-of-order cores are similar to Knights Landing [70]. Cores use SMT with up to 8 threads. Threads share the front-end and execution units, but have separate micro-op queues before the issue stage. The backend has two restricted execution ports: both ports can execute integer micro-ops, but floating-point micro-ops can run in port 0 only, and memory-access micro-ops can run in port 1 only. In-order cores are scoreboarded and stall-on-use, so even a single thread can have multiple memory requests in flight. Out-of-order cores feature a 36-entry issue buffer and a 72-entry ROB, both dynamically shared.

Benchmarks: We use a diverse set of ordered and unordered benchmarks. Table 4.2 details their provenance, input sets, and 1-core run-times on an in-order core. Most benchmarks have 1-core run-times of over one billion cycles.

We use eight ordered benchmarks. Six are the graph analytics (`bfs`, `sssp`, `astar`, `msf`), simulation (`des`), and database (`silo`) applications from the original Swarm

Cores	64 cores, 16 tiles, 2 GHz, x86-64 ISA, SMT with 1–8 threads
Frontend	8B-wide ifetch; 2-level bpred with 512×10-bit BHSRs + 1024×2-bit PHT; 16-entry per-thread micro-op queues
In-order backend	2-way issue, scoreboarded, stall-on-use, functional units as in Fig. 2-2, 16-entry load/store buffers
OoO backend	2-way issue/rename/dispatch/commit, 36-entry issue buffer, 72-entry ROB, 16-entry load/store buffers
L1 caches	16 KB, per-core, split D/I, 8-way, 2-cycle latency
L2 caches	256 KB, per-tile, 4 banks (64 KB/bank), 8-way, hashed, inclusive, 7-cycle latency
L3 cache	16 MB, shared, static NUCA [37] (1 MB slice/tile), 4 banks/tile, 16-way, hashed, inclusive, 9-cycle bank latency
Coherence	MESI, 64 B lines, in-cache directories
NoC	4 4×4 meshes; 192-bit links, X-Y routing, 1-cycle routers, 1-cycle links
Main mem	8 controllers at chip edges, 120-cycle latency, 25.6 GB/s per controller
Queues	128 task queue entries/core (8192 total), 32 commit queue entries/core (2048 total)
Instructions	5 cycles per `enqueue/dequeue/finish_task` instruction
Conflicts	2 Kbit 8-way Bloom filters, H_3 hash functions [15] Tile checks take 5 cycles (Bloom filters) + 1 cycle per timestamp compared in the commit queue
Commits	Tiles send updates to GVT arbiter every 200 cycles
Spills	Coalescers fire when a task queue is 87% full Coalescers spill up to 15 tasks each

Table 4.1: Configuration of the 64-core CMP.

	Source	**Input**	**1-core cycles**
bfs	PBFS [41]	hugetric-00020 [6, 19]	3.39 Bcycles
sssp	Galois [55]	East USA roads [1]	2.18 Bcycles
astar	[33]	Germany roads [51]	1.40 Bcycles
color	[27]	com-youtube [42]	1.08 Bcycles
msf	PBBS [9]	kron_g500-logn16 [6, 19]	0.74 Bcycles
des	Galois [55]	csaArray32	1.37 Bcycles
nocsim	GARNET [2]	16x16 mesh, tornado traffic	19.65 Bcycles
silo	[76]	TPC-C, 4 whs, 32 Ktxns	2.22 Bcycles
ssca2	STAMP [45]	-s15 -i1.0 -u1.0 -l6 -p6	11.13 Bcycles
vacation-l		-n2 -q90 -u98 -r1048576 -t262144	2.85 Bcycles
vacation-h		-n4 -q60 -u90 -r1048576 -t262144	3.86 Bcycles
kmeans-l		-m40 -n40 -i rand-n16384-d24-c16	7.81 Bcycles
kmeans-h		-m15 -n15 -i rand-n16384-d24-c16	3.10 Bcycles
genome		-g4096 -s48 -n1048576	2.06 Bcycles
intruder		-a10 -l64 -s32768	2.02 Bcycles
yada		-a15 -i ttimeu100000.2	2.79 Bcycles

Table 4.2: Benchmark information: source implementations, inputs, and execution time on a single in-order core, single-thread baseline system.

paper [33], and use the same inputs. The other two, **`color`** and **`nocsim`**, are from [32] and use the same inputs. **`color`** performs graph coloring using the largest-degree-first heuristic [82]. **`nocsim`** is a detailed NoC simulator derived from GARNET [2].

We use eight unordered, transactional memory benchmarks from STAMP [45]. We implement transactions with tasks of equal timestamp, so that they can commit in any order. As in prior work in transaction scheduling [4, 84], we break the original threaded code into tasks that can be scheduled asynchronously and generate children tasks as they find more work to do. The default "+" and "++" configurations are either too short in our largest system (512 threads), or too long to be simulated in reasonable time, respectively, so we use custom configurations that interpolate between the default ones.

We use all STAMP applications except **bayes** and **labyrinth**. Like Blake et al. [8], we observe that **bayes** has non-deterministic behavior that makes its runtime vary wildly, making comparisons across runs difficult. **labyrinth** consists of few very long transactions that conflict frequently and all but serialize execution. Hence, it does not make sense to run it on a 512-thread system (the whole program runs fewer transactions than the system has threads). We also observe that **intruder** and **yada** use software task scheduling data structures that limit their scalability. We refactor both applications to use Swarm's hardware task scheduling instead, which improves their scalability.

Metrics: We report average performance changes using *harmonic-mean* speedups.

On issue slot breakdowns (e.g., Figs. 2-1 and 2-3), we account for each stall reason in proportion to the number of threads it prevents from issuing. For example, if an issue slot cannot be used because 3 threads have *no ready* micro-ops and the remaining 5 have *no task*, *not ready* is charged for 3/8 of the slot, and *no task* for 5/8. If a thread uses the slot, stalled threads are not charged.

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

Chapter 5

SAM Evaluation

5.1 Multithreaded Scalability

Fig. 5-1 compares the performance and scalability of systems with 1 to 64 in-order cores, using three configurations: single-threaded cores, and 8-threaded SMT cores with the Round-Robin (RR) and SAM policies. As we scale the number of cores, we keep *per-core* L2/L3 sizes and queue capacities constant. This captures performance per unit area. Note that this causes some super-linear speedups because the larger shared L3 and hardware queues reduce memory pressure and task spills, respectively. Each line shows the speedup of a single configuration over the 1-core single-threaded system.

Overall, multithreading improves performance over the single-threaded configuration, by 2.41× with SAM and by 1.91× with RR on average. Over all benchmarks, SAM outperforms RR by 20% in harmonic speedup.

Four applications (**ssca2**, **vacation-l**, **vacation-h**, and **kmeans-l**) do not suffer from any multithreading pathology: they have negligible aborts and conflicts, and do not overload commit queues. Thus, they are insensitive to the issue policy—RR and SAM perform identically.

For all other applications, SAM consistently outperforms RR. SAM's benefits usually increase with the number of cores, as application parallelism becomes more scarce, and pathologies more frequent. SAM eliminates or ameliorates these pathologies. On these applications, SAM outperforms RR by 29% on average, and by up to 88% (**yada**).

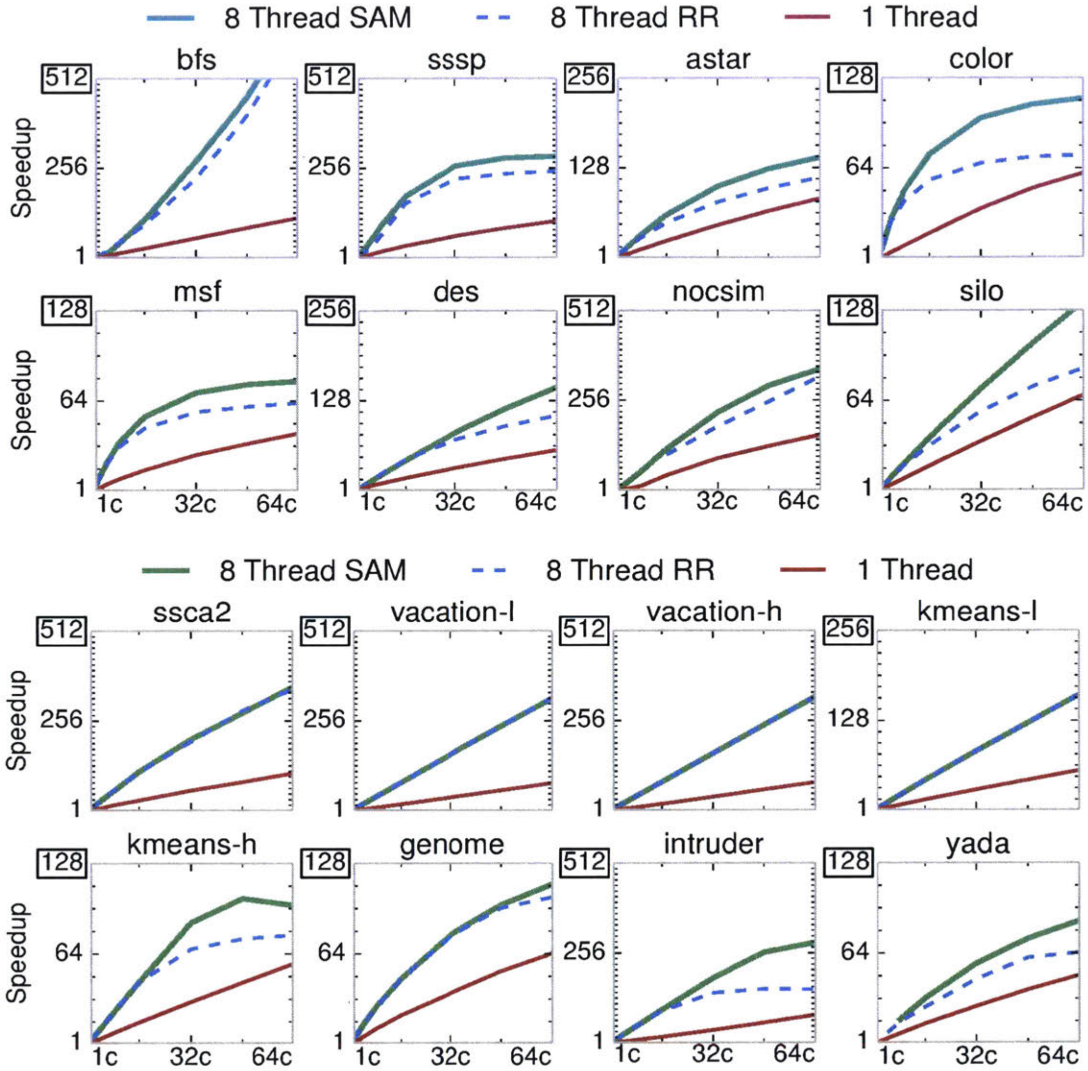


Figure 5-1: Performance of single-threaded cores and 8-threaded SMT cores with Round-Robin and SAM on (a) ordered and (b) STAMP benchmarks, as the system scales from 1 to 64 cores. Speedups are relative to the 1-core single-thread system.

5.2 Analysis of SAM for In-Order Cores

To gain more insights into the differences between SAM and RR, Fig. 5-2 reports the execution time and issue slot breakdown at 64 cores. Similar to Fig. 2-1, it shows how increasing the number of threads per core affects execution time. Each seven-bar group reports results for one application, using single-threaded cores as well as 2-, 4-, and 8-threaded cores with both RR and SAM. Results are normalized to those of single-threaded cores (lower bars are better).

Overall, increasing the number of threads per core has three dominant effects:

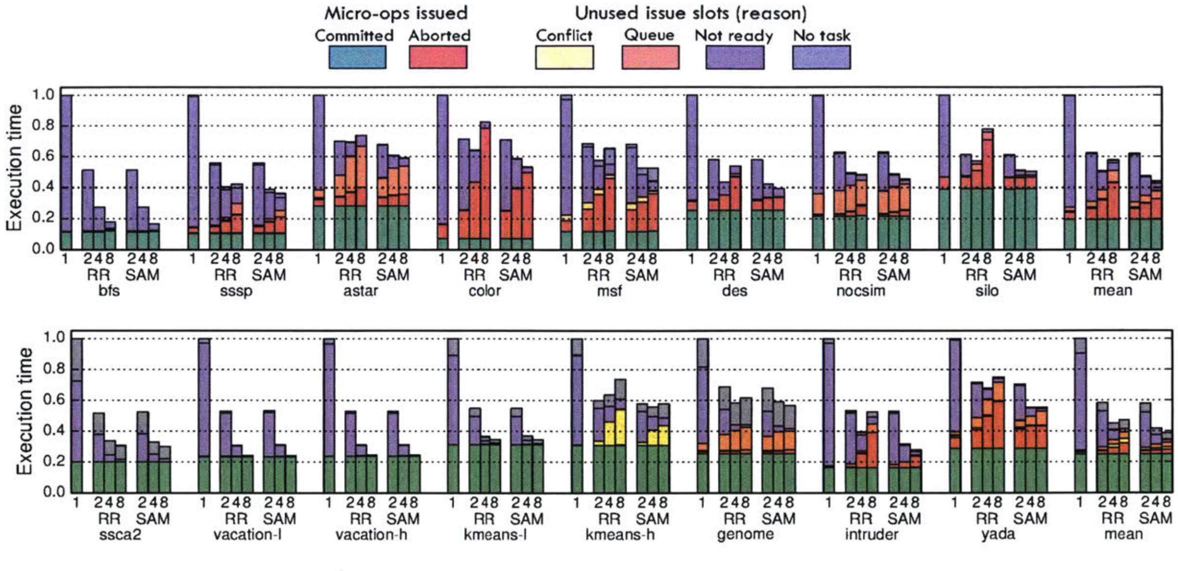


Figure 5-2: Execution times and breakdown of issue slots at 64 cores (in-order) for (a) ordered and (b) STAMP benchmarks, under a single-threaded configuration, and 2-, 4-, and 8-threaded configurations with Round-Robin and SAM (lower is better).

(i) not-ready stalls decrease, *(ii)* conflict stalls and issue slots lost to aborted tasks increase, and *(iii)* queue stalls increase. By prioritizing the execution of tasks that are more likely to commit, SAM mitigates the latter two factors and improves on RR. We analyze how these factors affect applications with different contention characteristics and speculation requirements.

RR and SAM perform equally well on applications without pathologies: Ordered `bfs` and unordered `ssca2`, `vacation-l`, `vacation-h`, and `kmeans-l` have plentiful parallelism but are memory-bound. With little contention, most time is spent issuing instructions from tasks that commit, or stalled on long-latency loads. RR and SAM perform equally well by reducing not-ready stalls. However, even eight threads per core cannot hide all memory latency in `bfs` and `ssca2`, and some stalls remain. At eight threads per core, these applications complete 2.9× (`kmeans-l`) to 5.6× (`bfs`) faster than with single-threaded cores.

SAM reduces wasted work and conflicts under contention: `color` has occasional data dependences among tasks, so adding threads increases aborts. With RR, aborts grow to the point of overwhelming the benefit of reduced stalls. However, since SAM prioritizes issues from tasks that are more likely to commit, it tempers the performance loss caused by aborted work. At eight threads per core, SAM is 55%

faster than RR on **color**, and 88% faster than with single-threaded cores. **msf**, **des**, and **silo** exhibit similar behavior.

The STAMP benchmarks **genome**, **intruder**, and **yada** also benefit from SAM. Though these applications are unordered, transactions inherit an order from the dynamic manifestation of dependences. Prioritization based on this order reduces wasted work by as much as 2.1× (**yada**).

On **kmeans-high**, conflict stalls, caused when the Wait-N-GoTM protocol detects a likely cyclic dependence, negate the reduction in not-ready stalls. SAM reduces the chance of such dependences by reducing the overlap of transactions.

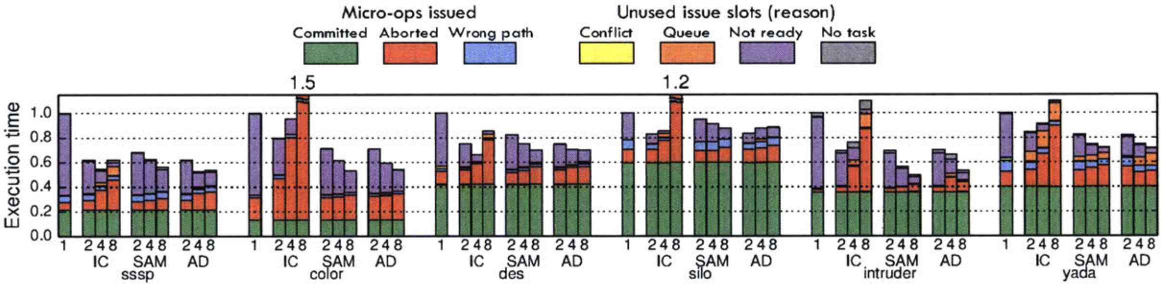


Figure 5-3: Execution times and breakdown of issue slots with 64 *out-of-order* cores for selected benchmarks, under a single-threaded configuration, and 2-, 4-, and 8-threaded configurations with ICount, SAM, and ADaptive policies.

SAM reduces queue stalls on applications that need a large speculation window: To find independent work, ordered applications may speculate so far ahead that they fill their commit and task queues, causing queue stalls. Queue stalls are significant in many ordered benchmarks and **astar** exemplifies this phenomenon. As we saw in Sec. 2.1, in **astar**, increasing threads per core with RR causes queue stalls to grow to the point of negating the benefits of reduced not-ready stalls. SAM reduces queue stalls by focusing execution resources on tasks with a lower virtual time, which must commit earlier. At eight threads per core, SAM is 25% faster than RR on **astar**, and 69% faster than the single-threaded configuration. **sssp** and **silo** exhibit similar effects; SAM improves their performance by reducing both queue stalls and aborted issue slots. **nocsim**'s queue stalls are significant, but do not grow beyond two threads per core; SAM helps **nocsim** by reducing aborted work, not queue stalls.

5.3 Analysis of SAM for Out-of-Order Cores

Fig. 2-5 in compares the average performance of ICount (IC) and SAM on an out-of-order core.[1] Overall, out-of-order cores are able to cover more stalls, so the performance benefits of multithreading are limited. However, a comparatively larger fraction of issue slots are wasted to aborts, hence the need for SAM is higher. On average, 8-threaded cores improve performance over single-threaded cores by 1.53× with SAM vs only 1.16× with ICount. Moreover, at 8 threads, SAM reduces wasted work by 81% over IC.

To understand these differences, Fig. 5-3 reports the execution time and issue slot breakdown for six representative applications. **`color`**, **`silo`**, **`intruder`**, and **`yada`** show that *aborts and conflict/queue stalls are the first-order concern in OoO cores.* With IC, cycles lost to these pathologies make these applications *slower* on 8-threaded cores than on single-threaded cores. By contrast, SAM keeps cycles lost to aborts and conflict/queue stalls nearly flat, outperforming IC by up to 2.9× (**`color`**). This happens even though IC reduces not-ready stalls and wrong-path execution more than SAM.

`sssp` shows how the adaptive policy can be beneficial. With 2 and 4 threads per core, IC's better pipeline utilization makes IC outperform SAM. With SAM, a single thread grabs most ROB entries, starving lower-priority threads. The adaptive policy detects this situation (aborts + wrong-path ¡ not-ready stalls) and opts for the higher pipeline efficiency of IC. **`des`** and **`silo`** show similar behavior.

Finally, **`intruder`** shows a case where the adaptive policy is suboptimal. With 4 threads per core, SAM has more not-ready stalls than IC but it more than makes up for it by reducing aborts. Therefore, the basic SAM policy is 36% faster than IC. However, the adaptive SAM policy, which by design tries to equalize aborts and stalls, attains a middle ground, where it is only 15% better than IC. Though they occur, these anomalies are very rare, and adaptive SAM nearly always matches the best of SAM and IC.

[1] We have also evaluated using RR instead of IC in OoO cores, but, like prior work, we find that IC is consistently better.

5.4 Case Study: Throttling

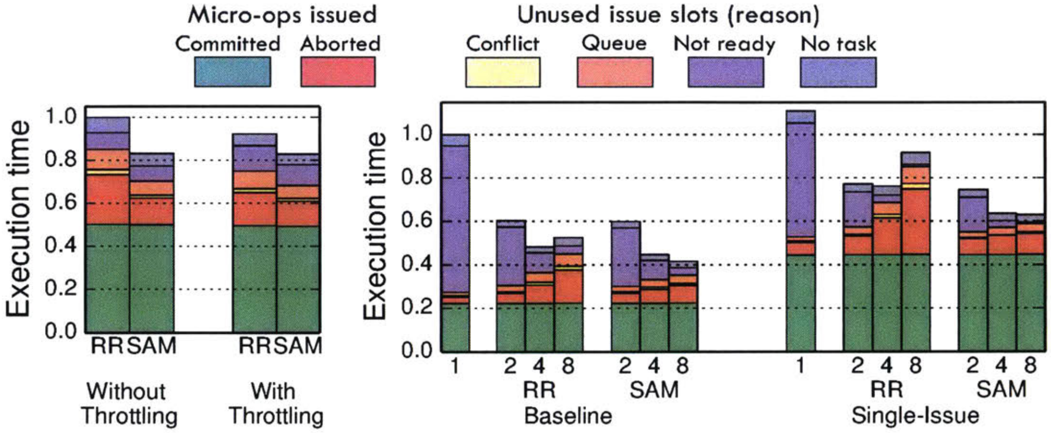
Throttling, i.e., limiting the amount of tasks executed in parallel, is a general strategy to cope with performance degradation caused when hardware parallelism exceeds application parallelism. Prior work has proposed dynamic throttling for non-speculative parallel programs [56, 72], as well as transactional schedulers that limit the number of concurrent transactions [4, 7, 8, 84], reacting to contention to reduce aborts. We show that adaptively limiting the number of active threads provides no benefit over SAM, and while throttling slightly improves RR, a large gap remains between RR and SAM.

We implement a simple throttler that builds on two insights. First, in all our applications, we observe there is a single thread count that performs best, and there are no other local maxima. Therefore, we use simple hill climbing to find the best number of threads per core. Second, many applications are either stable or change slowly over time. Therefore, we perform hill climbing as the application runs, incurring minimal cost.

Our throttler operates by periodically exploring nearby thread counts, and settles on the count that performs best. Since our applications are speculative, we use committed instructions per cycle as the performance metric (i.e., we do not consider executed instructions from tasks that later abort).

First, the throttler randomly chooses to either increase or decrease the number of active threads on every core in the system. If performance improves at the new thread count, the throttler continues changing the number of threads per core in the same direction, until it either reaches the minimum/maximum number of threads or performance degrades. If performance degrades, the throttler goes back to the previous thread count. This way, the throttler settles on the best-performing thread count among the explored ones. Each measurement interval is M cycles long, and the throttler stays at the new thread count for S cycles. We tune M ($50-500$K cycles) and S (250K$-$2.5M cycles) on a per-application basis to provide maximum benefit for each application.

RR with throttling yields marginal improvements, and a large gap with SAM remains: As shown in Fig. 5-4(a), throttling improves RR marginally, 8.4% on average at 8 threads per core. However, this is not sufficient to close the gap with



(a) Effect of throttling.

(b) Effect of issue-width.

Figure 5-4: Average execution time and issue slot breakdown for in-order 64-core (a) 8-threaded systems with RR and SAM, with and without throttling; and (b) systems with dual-issue (baseline) and single-issue cores.

SAM. Moreover, throttling does not improve SAM's performance. Throttling with RR only helps reduce aborts incurred at higher thread counts. In contrast, SAM reduces aborted instructions and queue stalls by prioritizing instructions from tasks that are likely to commit. Further, performance with throttling is sensitive to the throttler interval lengths—no single interval length performs best across all applications. Such careful parameter tuning makes throttling harder to apply than SAM.

In summary, throttling is inferior to SAM, as applying it to RR fails to capture most of SAM's improvements.

5.5 Sensitivity to Issue Width

Finally, Fig. 5-4(b) compares the behavior of a single-issue in-order core to the baseline 2-wide issue core. With one thread per core, the single-issue core performs 15% worse than the baseline. RR's performance degrades more rapidly beyond four threads, and 8-threaded RR cores are worse than 2-threaded cores. This happens because fewer threads are needed to avoid most stalls in the narrower pipeline. By contrast, SAM's performance does not degrade with thread count, although its benefits with increasing thread counts are reduced. Overall, this result shows that SAM avoids pathologies even when execution resources are more heavily contended.

Chapter 6

Prioritizing Main Memory Requests

SAM has shown that prioritization at the core is very useful. But cores are not the only shared resource: the memory system is also shared, and highly contended. Prioritization could be helpful here.

To explore this, we focus on memory controller, which is often the most contended resource in the memory system. Like SAM, deprioritizing requests from more speculative tasks inhibits their progress and less work is wasted if they were to ultimately abort. Unlike SAM, where each core makes a local decision on which thread to issue from, the memory controller is a global resource and can select among all threads in the system.

However, to our surprise, we find little benefit from prioritizing requests at the memory controller. When a task aborts, the data it fetched into the on-chip caches is usually reused when the task later re-executes and commits. Speculative execution works as a very effective prefetcher, and from the point of view of the memory controller requests served for aborted tasks are not wasted work. Therefore the benefits are minimal and we deem that prioritization in the memory system should not yield large benefits.

6.1 Prioritization at the Memory Controller

We use a conventional FR-FCFS (First Ready-First Come First Served) memory controller as our baseline. Here, priority is given to requests that hit in the open row-buffers of DRAM. If no request is a hit, requests are served in arrival order.

The controller maintains separate request queues for reads and writes. Since write request are never in the critical path of any task (they can only be generated due to LLC evictions), the controller prioritizes reads. When either the read queue becomes empty or when the write queue hits a high threshold, the bus is switched to write mode. Writes are serviced in a burst until the queue occupancy hits a lower threshold. This scheme minimizes the bus turnaround time.

We modify this controller to prioritize read requests by conflict resolution priority (i.e. virtual time in Swarm). The read queue is implemented with a hardware priority-queue indexed by virtual time and each memory access is tagged with the virtual time of the task that generated it. Write requests, which are not in the critical path, are not prioritized, and are scheduled according to FR-FCFS.

6.2 Methodology

We use the same simulation infrastructure as in Chapter 4 to explore prioritization at the memory controller. We model a DDR3-1600 memory with parameters given in Table 6.1.

DRAM	DDR3-1600, 4 ranks, 8 banks per rank
Burst Length	4 cycles
CAS latency	8 cycles
ACT to CAS	8 cycles
RD to PRE	4 cycles
PRE to ACT	8 cycles
ACT to ACT	4 cycles
ACT to PRE	24 cycles
WR to RD	4 cycles
WR to PRE	8 cycles
Queue Sizes	64 entry read queue, 64 entry write queue
Write Thresholds	48 high threshold, 16 low threshold

Table 6.1: Configuration of DRAM and the Memory Controller.

The memory controller scheduling scheme would not make a difference unless there are multiple requests competing for the bus at the same time. To maximize the amount of such requests, we run all experiments on a 64-core, 8-threads/core system, with memory bandwidth constrained to 12.6 GB/s. This is a very constrained amount of bandwidth for a 64-core chip. The reason is that we want the memory controller to be highly contended in this experiment to maximize the impact of the scheduling policy. We expect real systems to have more memory bandwidth.

6.3 Results and Discussion

Fig. 6-1 compares how the Virtual-Time prioritized (VT) memory controller performs against FR-FCFS. We leave out insensitive applications: those with negligible aborts and those which do not overload the commit queues from this graph.

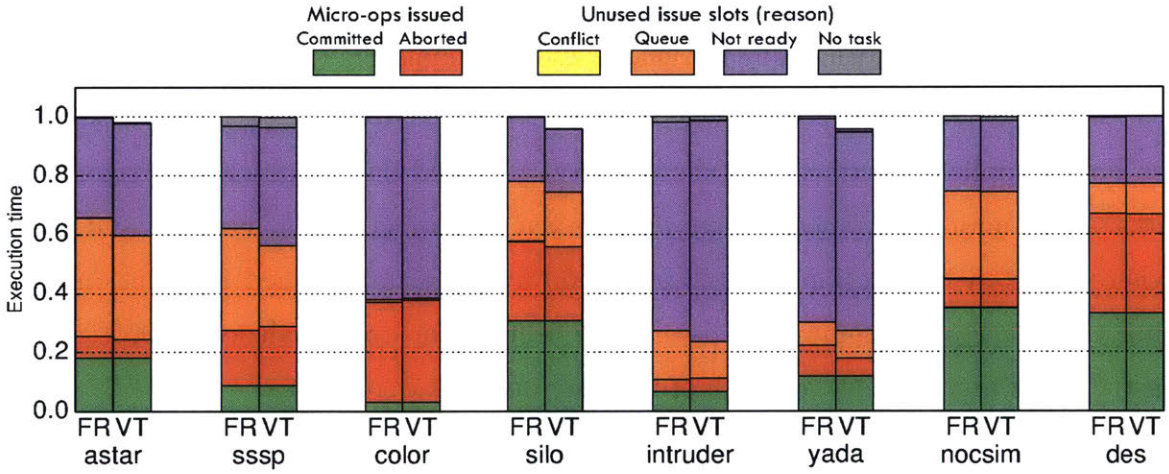


Figure 6-1: Execution times and breakdown of issue slots for selected benchmarks under FR-FCFS and Virtual Time memory controller scheduling schemes, using 64 in-order cores with 8 threads/core.

Surprisingly, even with this limited bandwidth, we find that no application shows any significant benefit. `silo` and `yada` show about a 5% benefit. Despite having significant aborts, none of the applications show any reduction when the memory controller is prioritizing accesses from tasks that are more likely to commit.

Fig. 6-2 provides more insights for this behavior, by showing a breakdown of how memory bandwidth is utilized. We distinguish between memory reads and writes, and reads are further broken down according to whether they were useful or wasted. We

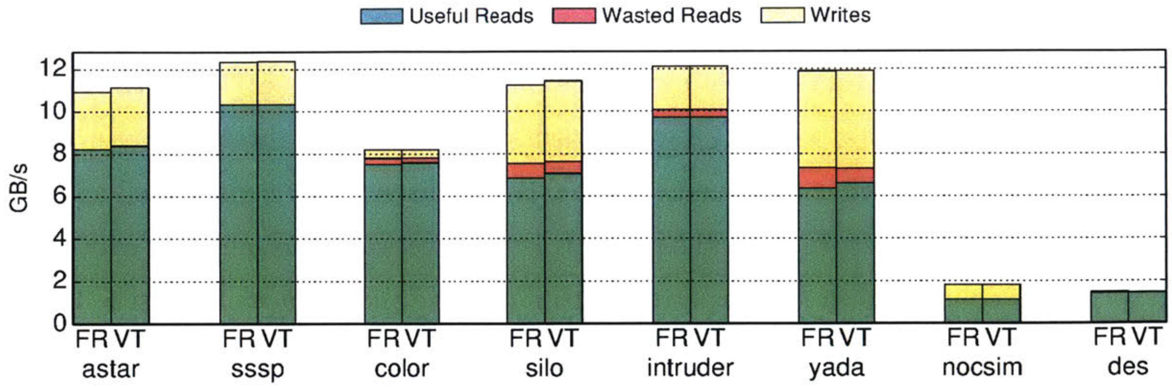


Figure 6-2: Memory bandwidth utilization and breakdown of memory accesses for selected benchmarks under FR-FCFS and Virtual Time memory controller scheduling schemes.

identify a read request as "useful" if the line is ever accessed by a task that commits. If all tasks that accessed the line ultimately abort, we mark the memory request that initially brought in the line to be "wasted".

Two applications, **nocsim** and **des**, are not memory bandwidth bound, even in this restricted bandwidth configuration. These are simulators that have small working sets that fit in the LLC. Under such circumstances, prioritization has little effect.

For all applications (including the above two), we find that despite having significant aborts, the majority of memory requests served are in fact useful. For **des** and **sssp**, where aborted instructions are 30% and 60% of the total issued instructions, we find that nearly all memory accesses were useful. For **color**, although 90% of the instructions are aborted, only 5% of memory accesses were wasted.

These results show that speculative execution works as a very effective prefetching mechanism. Speculatively running tasks far ahead in program order brings the data they access into the on chip caches. Even if the original task aborts, the data is used by another task that commits. This typically, but not always, happens when the aborted task is later re-executed. Hence, from the point of view of the memory controller, requests served for aborted tasks are not wasted work.

In fact, this is the same principle that runahead execution [48], a technique where instructions are executed speculatively to generate accurate prefetches in out-of-order cores, takes advantage of. We encounter the same effect with task-level speculation.

However, **yada** and **silo** show that prioritization can benefit when wasted memory accesses are more prevalent. Such accesses occur, for example, when either a speculative task is not rerun or accesses a different region of memory when rerun. Deprioritizing more-speculative tasks prevents them from making further progress and leads to a reduction in wasted work.

Several applications (e.g. **sssp**, **intruder**) show a reduction in queue stalls. Like in SAM, prioritizing tasks with a lower virtual time will ensure a better utilization of the commit queue resources. Unlike SAM, this reduction does not lead to overall benefits, since performance is ultimately limited by memory bandwidth. (We also explored systems with higher bandwidth, but prioritization does not make a difference when the memory controller is not contended.)

Chapter 7

Additional Related Work

7.1 Multithreading in Speculative Parallelism

IMT [54] is perhaps the closest proposal to SAM. For a multithreaded single-core TLS system, IMT prioritizes the sole non-speculative thread when inter-thread dependences are frequent. In contrast, SAM derives core-local priorities for all cores and threads in the system. While IMT is sensible in a 1-core system, on the 512-thread system we evaluate, IMT would have negligible impact by prioritizing the one thread (system-wide) that runs the single non-speculative task.

Other work has supported speculative parallelization on SMT cores, first in the context of TLS [3, 52, 80], and more recently on HTM [57]. These proposals focus on tailoring the versioning and conflict detection mechanisms to SMT cores. However, these systems use conventional multithreading policies, such as round-robin or ICount [77]. By contrast, SAM shows that coordinating issue and conflict resolution priorities makes speculation much more efficient.

Recent work has implemented HTM for GPUs [21, 22], which have heavily multithreaded cores. Like the above designs, this work focuses on tailoring speculative mechanisms to the characteristics of GPUs, to cope with their large numbers of threads and exploit their data-parallel nature. These techniques also use conventional multithreading policies.

7.2 Prioritization in Non-Speculative Systems

Prior work has proposed SMT prioritization policies for parallel programs. Tullsen et al. [78] propose fine-grain synchronization techniques to accelerate lock-based programs. Cai et al. [14] and Boneti et al. [11, 12] use SMT priorities to address work imbalance in barrier-based programs. Beyond SMT, ACS [75] and BIS [35] accelerate critical sections and other bottlenecks in multithreaded programs by scheduling them in fast cores on a heterogeneous system. These prioritization techniques are useful to accelerate non-speculative synchronization constructs, but not speculative parallelism, where all synchronization among tasks is implicit.

Prior work has also proposed many GPU thread (i.e., warp) prioritization schemes [36, 43, 49, 65, 68]. These schemes mainly seek to improve locality by limiting the number of threads that are interleaved at fine granularity. Locality is the overriding concern in GPUs because they are heavily threaded and have very little on-chip storage per thread. However, issue policies have a minor effect on locality for the number of threads per core we consider.

Finally, some SMT systems expose issue priorities to software [11, 23]. While our SAM implementation controls priorities in hardware, software TM or TLS systems could use this support to implement SAM.

7.3 Scheduling in Memory Controller

Thread-agnostic memory scheduling algorithms [29, 64, 69, 85, 86] seek to maximize DRAM throughput. FR-FCFS [64], which seeks to maximize row hits, is the most common in existing processors. For multiprogrammed workloads, where different threads have completely different characteristics and different memory level parallelism, these policies can cause unfairness. Prior techniques [47, 60] have sought to solve this unfairness, but they sometimes sacrifice system throughput [50]. TCM [39] achieves the best of both by prioritizing latency-sensitive threads over bandwidth-sensitive ones. However, all these proposals target systems running multiprogrammed workloads, not a single multithreaded application as we do.

Chapter 8

Conclusion

This thesis has explored how throughput-oriented architectures can be optimized for speculative parallelism. Naively adding hardware support for speculative parallelism in these architectures leads to a severe pathology: aborted work consumes scarce resources and hurts the throughput of useful work.

To mitigate this pathology, we proposed that throughput-oriented architectures should prioritize tasks according to how speculative they are. We identified two on-chip resources where this prioritization is most likely to be beneficial.

First, we presented Speculation-Aware Multithreading (SAM), a simple technique to prioritize instructions at the core pipeline by aligning instruction dispatch with conflict resolution priorities. By focusing execution resources on likely-to-commit tasks, SAM reduces aborts and conflicts; and since these tasks commit earlier, SAM also makes more effective use of speculation resources.

SAM improves the performance benefit of multithreaded cores on speculative programs. On a 64-core system with 2-wide issue in-order SMT cores, 8-threaded cores outperform single-threaded ones by 2.41× on average with SAM, vs. by 1.91× with round-robin. SAM also reduces wasted work by 43%. With out-of-order execution, 8-threaded cores outperform single-threaded cores by 1.53× with SAM vs only 1.16× with ICount, and SAM reduces wasted work by 81%.

Second, we investigated whether prioritizing requests at the memory controller according to their conflict resolution priorities can provide performance benefits. However, unlike at the core, we found little performance benefit from prioritizing requests at the memory controller. The reason is that speculative execution works as a very effective prefetching mechanism, and most requests, even those from tasks that are ultimately aborted, do end up being useful.

These insights would help computer architects understand the trade-offs involved in designing future computing systems. With the end of technology scaling, leveraging parallelism has become the most effective way of making applications run even faster. Throughput-oriented architectures were traditionally used to run massively parallel applications, yet its interaction with hardware support for speculation, which expands the range of applications that can be parallelized, remained unexplored. As we show in this thesis, prioritizing at the core pipeline is a simple but useful technique that should be incorporated into multithreaded systems with support for speculative parallelism, whereas prioritizing memory requests is not necessary.

Bibliography

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

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

[3] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *MICRO-31*, 1998.

[4] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *HiPEAC*, 2009.

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

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

[7] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *MICRO-42*, 2009.

[8] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom filter guided transaction scheduling," in *MICRO-44*, 2011.

[9] G. Blelloch, J. Fineman, P. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *PPoPP*, 2012.

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

[11] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero, "Software-controlled priority characterization of POWER5 processor," in *ISCA-35*, 2008.

[12] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero, "A dynamic scheduler for balancing HPC applications," in *SC08*, 2008.

[13] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Linux Symposium*, 2012.

[14] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *PACT-17*, 2008.

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

[16] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: bulk enforcement of sequential consistency," in *ISCA-34*, 2007.

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

[18] J. D. Davis, J. Laudon, and K. Olukotun, "Maximizing CMP throughput with mediocre cores," in *PACT-14*, 2005.

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

[20] J. Emer, "EV8: the post-ultimate alpha," *Keynote at PACT*, 2001.

[21] W. W. Fung and T. M. Aamodt, "Energy efficient GPU transactional memory via space-time optimizations," in *MICRO-46*, 2013.

[22] W. W. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for GPU architectures," in *MICRO-44*, 2011.

[23] R. Golla and P. Jordan, "T4: A highly threaded server-on-a-chip with native support for heterogeneous computing," in *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011.

[24] L. Gwennap, "ThunderX rattles server market," *Microprocessor Report*, vol. 29, no. 6, 2014.

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

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

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

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

[29] I. Hur and C. Lin, "Adaptive history-based memory schedulers," in *MICRO-37*, 2004.

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

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

[32] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez, "Data-Centric Execution of Speculative Parallel Programs," in *MICRO-49*, 2016.

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

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

[35] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS-XVII*, 2012.

[36] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance," in *ASPLOS-XVIII*, 2013.

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

[38] H. Kim, R. W. Vuduc, S. S. Baghsorkhi, J. Choi, and W. mei W. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU).* Morgan & Claypool Publishers, 2012.

[39] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. of the 43rd intl. symp. on Microarchitecture*, 2010.

[40] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 25, no. 2, 2005.

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

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

[43] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *HPCA-21*, 2015.

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

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

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

[47] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. of the 40th intl. symp. on Microarchitecture*, 2007.

[48] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *HPCA-9*, 2003.

[49] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *MICRO-44*, 2011.

[50] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair queuing memory systems," in *Proc. of the 39th intl. symp. on Microarchitecture*, 2006.

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

[52] V. Packirisamy, Y. Luo, W.-L. Hung, A. Zhai, P.-C. Yew, and T.-F. Ngai, "Efficiency of thread-level speculation in SMT and CMP architectures-performance, power and thermal perspective," in *ICCD*, 2008.

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

[54] I. Park, B. Falsafi, and T. N. Vijaykumar, "Implicitly-multithreaded processors," in *ISCA-30*, 2003.

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

[56] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *IISWC*, 2008.

[57] X. Qian, B. Sahelices, and J. Torrellas, "BulkSMT: Designing SMT processors for atomic-block execution," in *HPCA-18*, 2012.

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

[59] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors," in *PACT-12*, 2003.

[60] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *PACT-16*, 2007.

[61] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *ISCA-32*, 2005.

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

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

[64] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *ISCA-27*, 2000.

[65] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO-45*, 2012.

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

[67] W. N. Scherer III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, 2005.

[68] A. Sethia, D. A. Jamshidi, and S. Mahlke, "Mascar: Speeding up gpu warps by reducing memory pitstops," in *HPCA-21*, 2015.

[69] J. Shao and B. T. Davis, "A burst scheduling access reordering mechanism," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, Feb 2007, pp. 285–294.

[70] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, 2016.

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

[72] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *PLDI*, 2014.

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

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

[75] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *ASPLOS-XIV*, 2009.

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

[77] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *ISCA-23*, 1996.

[78] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fine-grained synchronization on a simultaneous multithreading processor," in *HPCA-5*, 1999.

[79] G. Voskuilen, F. Ahmad, and T. Vijaykumar, "TimeTraveler: Exploiting acyclic races for optimizing memory race recording," in *ISCA-37*, 2010.

[80] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded multiple path execution," in *ISCA-25*, 1998.

[81] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *PACT-21*, 2012.

[82] D. J. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

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

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

[85] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee, "The impulse memory controller," *IEEE Trans. Comput.*, vol. 50, no. 11, pp. 1117–1132, Nov. 2001.

[86] W. Zuravleff and T. Robinson, "Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order," May 13 1997, US Patent 5,630,096.