Hardware Multithreaded Transactions: Enabling Speculative Multithreaded Pipeline Parallelization For Complex Programs

Jordan Fix

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: David I. August

January 2020

Abstract

Speculation with transactional memory systems helps programmers and compilers produce profitable thread-level parallel programs. Prior work shows that supporting transactions that can span multiple threads, rather than requiring transactions be contained within a single thread, enables new types of speculative parallelization techniques for both programmers and parallelizing compilers.

Unfortunately, software support for multi-threaded transactions (MTXs) comes with significant additional inter-thread communication overhead for speculation validation. This overhead can make otherwise good parallelization unprofitable for programs with sizeable read and write sets.

Some programs using these prior software MTXs overcame this problem through significant efforts by expert programmers to minimize these sets and optimize communication, capabilities which compiler technology has been unable to achieve to date. Instead, this dissertation makes speculative parallelization less laborious and more feasible through low-overhead speculation validation, presenting the first complete design, implementation, and evaluation of hardware MTXs.

Even with maximal speculation validation of every load and store inside transactions of tens to hundreds of millions of instructions, profitable parallelization of complex programs can be achieved. Across 8 benchmarks, this system achieves a geomean speedup of 104% over sequential execution on a multicore machine with 4 cores, with modest increases in power and energy consumption. This work could be used as a building block to enable more realistic automatic parallelization of complex programs, by providing low-overhead, long-running, resilient transactions that support a diverse set of parallel paradigm options.

Acknowledgments

I would first like to thank my advisor Professor David I. August for the many years of support he provided me. Through so many years of research guidance and long nights of paper and proposal deadlines, he has helped me develop into a much better researcher and writer. He helped inspire a variety of different projects I was fortunate to be a part of during my many years in Princeton. Additionally, the culture he set in the Liberty Research Group created a spirit of friendship, selflessness, and camaraderie that made my time in graduate school so much more fulfilling.

I very much appreciate the time he and Dr. David Callahan spent reading and providing feedback and guidance for this dissertation. Additionally, I want to thank Professor Andrew Appel, Professor Margaret Martonosi, and Professor David Wentzlaff for serving as examiners on my committee, and for providing early feedback on how I could make my research stronger.

Next, the members of the Liberty Research Group made possible all of my work and growth as a computer scientist from my very first day in Princeton. They are all deserving of my gratitude – Arun Raman, Tom Jablin, Hanjun Kim, Prakash Prabhu, Nick Johnson, Feng Liu, Matt Zoufaly, Stephen Beard, Taewook Oh, Soumyadeep Ghosh, Heejin Ahn, Hansen Zhang, Nayana Prasad Nagendra, Sergiy Popovych, Sotiris Apostolakis, and Sophie Qiu. The many long hours of writing and rewriting papers, of coding and running experiments, of perfecting every pixel on every slide of presentations, and of complaining about graduate school with all of you made my years in Princeton something I will never forget, for better or for worse.

I especially want to thank Nayana, Sotiris, Hansen, and Deep for helping with different parts of the project behind this dissertation, from writing code to proofreading papers and nit-picking presentations. I could not have gotten past the finish line without your help, and your years of help are greatly appreciated. Additionally, I would like to thank Neil Vachharajani for blazing the very beginning of this project. His work provided a starting

point for me that was an invaluable source of inspiration.

I also want to thank other members of the Computer Science department, including Nicki Gotsis and Melissa Lawton, the graduate coordinators during my time at Princeton, who helped me navigate the many requirements during my generals, pre-FPO and FPO.

Outside of my academic time in Princeton, I would not have survived without 6 Madison. My roommates Aaron, Ed, and Pawel made our many years so much fun. I enjoyed our many ski, surf, and beach trips, BBQs and cocktails alongside cornhole in the backyard, and so much more. To everyone else who were regular faces in our house and on our many outings, including Tanya, Alex, Julia, Daniel, Harvey, and so many others, thank you for listening to me blow off steam in between all of the fun.

To my wife Rotem, thank you for pushing me to finish this dissertation. I am not sure I would have gotten to the end without you. I so very appreciate the support you have provided across the past few years. My final years in Princeton were made so much more amazing thanks to you, whether in the beginning while taking many back and forth trips to Philly, or Skyping across the world with you.

Lastly, I want to thank my parents and sister, who were also instrumental in pushing me not just to finish graduate school but to start in the first place. I would never have gotten through all of these years of education without your advice and support, and it will always be appreciated.

I appreciate all of the support I received while in school. My time at Princeton and the research in this dissertation was funded by a variety of sources, including: “XPS: Exploiting Parallel & Scalabilty” (NSF Award DMS-1439085); “SI2-SSI: Software Infrastructure for Sustained Innovation - Scientific Software Integration” (NSF Award OCI-1047879); and “SPARCHS: Symbiotic, Polymorphic, Automatic, Resilient, Clean-slate, Host Security” (DARPA Award FA8750-10-2-0253).

For my parents, who supported me through too many years of education.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Due to fundamental constraints on power usage and heat dissipation, microprocessor manufacturers have resorted to multicore processors with multiple individual processing cores. However, multicore processors do not improve sequential program performance; programs must be modified to incorporate thread-level parallelism (TLP) to take advantage of these parallel resources, whether through data or task parallelism.

Static parallelization that conservatively respects all potential dependences has achieved success in some domains, such as streaming applications [2], MapReduce applications [3], and embarrassingly parallel DOALL [4] tasks such as matrix multiplication. For example, scientific codes are often vector-based with affine accesses and are therefore trivially thread-level parallelizable in a DOALL fashion, where each iteration of their hottest loops is fully independent and can be executed in parallel.

In order to extract other forms of TLP from more complex programs, DOACROSS [4] and pipeline parallelism-based techniques such as Decoupled Software Pipelining (DSWP) [5, 6, 7] have been invented. However, modern software is often so complex that it is hard to statically eliminate the possibility of dependence violations through accesses to shared data, even if such violations cannot manifest during execution. It is often hard or impossible for a compiler or programmer to determine if these techniques can be applied to a program. Ad-

ditionally, there may be opportunities for parallelization that are limited due to dependences that may manifest but rarely do. Thus, equivalent success has not been found for general purpose, complex programs, such as those with irregular pointer-chasing data structures. Past works have shown that consumer grade systems continue to be underutilized; even 4 cores appears to be over-provisioning for most applications [8, 9, 10].

To allow for more aggressive TLP extraction on these programs without explicit dependence synchronization via locks or communication, speculative execution and transactional memory (TM) systems have been explored. TM systems allow for loads and stores inside a critical region to atomically execute or roll back together. This atomic unit of work is called a transaction. Multiple transactions can speculatively execute in parallel safely; if any transactions conflict with each other then the TM system will roll back the transactions.

Almost all prior TM systems depend upon either DOALL or DOACROSS style parallelization. DOALL is not widely applicable outside of scientific codes. Additionally, DOACROSS [4] performance depends upon the inter-core latency of the system, as it requires that loop carried dependences be communicated to other cores for every iteration.

As an alternative, speculative parallel pipeline techniques such as speculative DSWP can be used [1, 11, 12, 13]. These past works have found that speculatively pipeline parallelizing a program often has better performance than other parallelization schemes using traditional Thread Level Speculation (TLS).

Unfortunately, most TM systems used for TLS [14, 15, 16, 17, 18] do not provide sufficient support for speculative pipelined parallelism, which split individual transactions across multiple pipelined threads. Speculative pipeline parallelism requires TM systems that support *multithreaded transactions* (MTXs), wherein multiple threads can collaborate on a single transaction that can atomically commit or rollback.

Some software TM (STM) systems [12, 13] have been developed to include MTX support, allowing speculative pipeline parallelism techniques to be used on commodity hardware. However, these systems suffer from high runtime overheads, which can curtail the

performance of what would otherwise be well-performing parallel programs. Most troublesome is the overhead from communication of large read and write sets for transaction validation, which can be prohibitively costly [19].

Thus, to be useful for complex programs, these STM systems must limit the amount of speculation validation performed. This requires laborious expert manual transformation to avoid these overheads and achieve speedup. Without further advances in compiler analyses to eliminate the need for significant amounts of validation checks, these STM systems are not ideal for automatic speculative parallelization of complex programs.

Prior work [20] examined the importance of static dependence analysis in a speculative automatic parallelizing compiler for simple programs with affine accesses such as matrix multiplication. Scalable speedup turned into significant slowdown when using weaker dependence analysis due to increased speculation validation overhead. Even with the strongest modern static analyses, more complex programs have not been profitably parallelized due to required speculation validation.

These aliasing issues cannot be ignored; they are often hard or impossible to reason about, making speculation an attractive alternative solution to statically determining them. And even if a legitimate may-alias relationship between two pointers or addresses is statically determined, that aliasing may never manifest, resulting in a missed opportunity for parallelization. Therefore, supporting low-overhead alias speculation support is essential.

Thus, a TM system with low-overhead speculation validation is essential to achieve automatic parallelization of complex programs and more widespread use of speculative parallel execution. Hardware TM (HTM) systems can provide this low-overhead speculation validation. However, no HTM systems with MTX have been comprehensively explored. Additionally, existing HTM systems do not provide sufficient support for long-running and complex programs that often require long-running and complex transactions for speculative execution.

For example, Intel HTM in Haswell processors are limited to fitting their entire write

sets inside the L1 [21], meaning transactions cannot speculatively write very much data without forcing an abort. Additionally, many modern processors aggressively execute loads out of order, some of which end up squashed due to branch misprediction. Without extra care being taken, squashing loads that were also marked as speculative by the TM system could result in unnecessary transaction aborts, degrading performance.

This dissertation overcomes the limitations of prior TM systems, presenting the first complete design, implementation, and evaluation of a TM system with support for hardware multi-threaded transactions (HMTXs) [22]. In this system:

- Multiple threads can collaborate on a single transaction, with uncommitted memory modifications visible to all threads working on the transaction, and with the ability for these modifications (potentially spread across many caches) to atomically commit together.
- Multiple transactions can execute on a single core without requiring any of them to commit or abort, allowing for a thread to finish work on one transaction and begin on another without interfering with the first (which may still be uncommitted).

Additionally, in order to provide for long-running and complex transactions, in this system:

- Transactions are resilient; they are novel by avoiding false misspeculation due to branch misprediction, supporting large read and write sets, and allowing for interrupt and exception handling.
- A lazy commit scheme is used, efficiently processing large read and write sets (up to tens of megabytes of data in the evaluated benchmarks).

The combination of these features allows for the HMTX system to achieve profitable parallelization of complex, long-running programs with large amounts of speculation validation. This overcomes a large barrier to achieving automatic parallelization, and makes it easier for compilers and programmers alike to create well performing parallel programs.

This dissertation presents 8 benchmarks (7 from the SPEC benchmark suite [23, 24],

and 1 from MiBench [25]) that are speculatively parallelized with the maximal possible amount of speculation validation, i.e. conservatively adding *every load and store inside a transaction* (often made of up of tens to hundreds of millions of instructions) to the read and write set. Despite such large amounts of validation, this system achieves a geomean speedup of 104% over sequential execution on a multicore machine with 4 cores, exhibiting the limits to which transactional memory can be used while sustaining good parallel performance, and with modest increases in power and energy consumption.

Chapter 2

Background and Motivation

This section motivates the need for MTX support. Non-speculative TLP techniques are detailed in Chapter 2.1. Speculative versions of these techniques are discussed in Chapter 2.2. Current state-of-the-art TM systems with MTX support are described in Chapter 2.3.

2.1 Thread Level Parallelization Techniques

Techniques such as DOALL, DOACROSS [4, 26], and pipeline parallelization [5, 6, 7] have been used in order to better leverage multicore architectures via TLP. In DOALL, each iteration of a loop is fully independent of the others and therefore each iteration can be executed in parallel. This is mostly applicable only to scientific programs that perform affine operations on regular data structures.

DOACROSS can extract parallelism from more complex loops with loop-carried dependences. Consider a linked list in which some work function is performed on each node. Each iteration needs to know the previous iteration's node to find its own node, which means this dependence is loop-carried, and DOALL is therefore inapplicable.

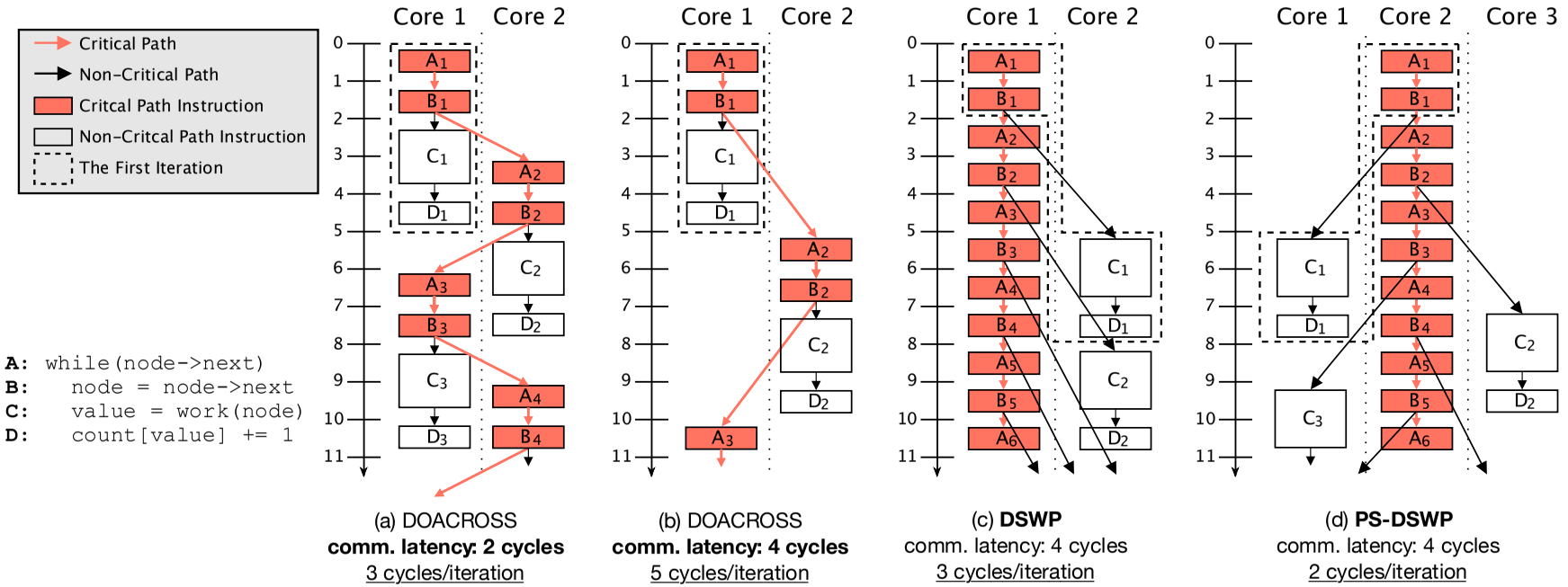


Figure 2.1: Execution timing diagram of the first iterations of a loop for DOACROSS, DSWP, and PS-DSWP with different inter-core latencies.

DOACROSS parallelizes this program in the following fashion. Core c_1 finds the first node n_1 (seen in code line/visualized as box `B`), and then sends the next node n_2 to core c_2. c_1 continues processing n_1, calling the work function; meanwhile, c_2 can start processing n_2 in parallel, and repeat this process by passing n_3 to another core. This execution model can be seen in Figure 2.1(a) and (b).

This loop could also be pipeline parallelized, for example by using Decoupled Software Pipelining (DSWP)[5], seen in Figure 2.1(c). Using DSWP, the work of each iteration is separated into a pipeline across multiple cores. In a DSWP parallelized version of the previous linked list example, core c_1 would iterate on finding every location n_i in the linked list, while sending these locations to c_2 for it to separately process the node by calling the work function.

Parallel-Stage DSWP (PS-DSWP) [6, 27] recognizes that the resulting work in the second stage of the pipeline can now be done in a DOALL fashion. This makes PS-DSWP more scalable than DSWP, performing much better than DSWP or DOACROSS. This can be visualized in Figure 2.1(d).

DOACROSS performance depends upon the inter-core latency of the system, because the loop carried dependence must be communicated between cores for every iteration. Meanwhile, pipeline parallelization techniques like DSWP and PS-DSWP are insensitive to inter-core latency, and only pay this price at the start of execution. Past works have found that DSWP style parallelism and its variants often have better performance than DOACROSS [12, 13, 1]. Figure 2.1 shows that DOACROSS and DSWP could only profitably make use of two cores; meanwhile, PS-DSWP can use many more cores.

2.2 Speculation

Due to the limits of static analysis, compilers and programmers often find it hard or impossible to make use of profitable parallelization opportunities. This is due to dependences

between would-be threads that rarely or never manifest, whose absence would allow for profitable parallelization. For example, there may be memory accesses into hard to analyze structures which prevent parallelization, requiring expensive synchronization or communication to ensure a correctly working parallel version of a program.

Additionally, there may be other inhibitors of parallelization based on control flow, e.g. error condition checks in a hot loop could prevent parallelization of a loop due to possible side exits from the loop. These inhibitors may be input dependent. For example, even if static analysis determines that two pointers into such structures may alias, this relationship may not manifest given program inputs and therefore may unnecessarily inhibit profitable parallelization. Even if these speculative assumptions about programs are not input dependent, they are often hard or impossible to prove, especially via a compiler's static analyses.

To overcome inhibiting problems such as hard to analyze structures or unlikely control flow, speculative parallelization is an attractive solution, allowing for optimistic parallel execution. If any speculative assumption is incorrect at runtime, misspeculation will be detected and the program state will roll back to a previously committed, valid state, undoing any potentially harmful effects.

Even if inhibitors of parallelization are input dependent, speculating them away can still be done highly confidently, for example based on profiling the program. Still, validation must be performed even if the inhibitors never manifest and trigger misspeculation. This means that low overhead speculative validation support is very important even if speculative parallelization is done with high confidence.

Many past TM systems provide low overhead validation support for speculative DOALL and DOACROSS via thread-level speculation (TLS) [14, 15, 16, 17, 28]. However, all past TLS systems are insufficient for speculative DSWP, which has been shown to often have better applicability and/or performance than speculative DOALL and DOACROSS [1, 11, 12, 13].

Speculative-DSWP requires multi-threaded transactions (MTXs), wherein transactions can span multiple threads. In DSWP, each iteration (wrapped in a transaction) is equivalent to a loop iteration, and each loop iteration is executed across multiple pipelined stages, which are executed on different threads. This means that each transaction is split across multiple pipelined threads[1] , as seen in Figure 2.1 (c) and (d). Therefore, when a thread executing the first stage of the pipeline makes some speculative modifications to memory as part of a transaction, those modifications should be visible to some other thread executing the second stage of the pipeline when continuing with execution of that same transaction, even though that transaction has not yet committed. Additionally, all speculative modifications that are part of a single transaction should atomically be committed at once, even though they are executed by different threads that are executing different pipeline stages.

2.3 Past Multithreaded Transaction Proposals

Vachharajani [1] described the general concept of multi-threaded transactions (MTXs). The proposal gives each MTX a *version ID* (VID). Speculative memory accesses from an MTX mark memory they touch with their VID. Versioning memory allows for key properties (described in detail in Chapter 3) which are requirements of an MTX system. While Vachharajani described an initial design for a hardware TM system with MTXs, no detailed implementation or evaluation was ever published. Additionally, parts of the design were unrealistic or left incomplete (Chapter 9).

Later MTX proposals opted for software based TM systems to allow for speculative DSWP execution on commodity hardware [12, 13, 29, 30]. These systems follow in the same path as Vachharajani, assigning a VID to each transaction. They provide for multiple

[1]Using DSWP's pipeline partitioning algorithm, instructions inside a parallelized loop's body are not necessarily kept in order with respect to the original program when partitioned across different pipeline stages. When combined with commonly used control flow speculation, it is insufficient to use traditional TLS transactions for each stage's individual portion of each iteration (e.g. in Figure 2.1(c) and (d), putting A_1 and B_1 in a separate TLS transaction from C_1 and D_1.)

versions of memory by forking the main process multiple times for each parallel worker. There is an additional process called the *commit process*, which contains and manages the committed non-speculative state. This design allows the other forked, speculative worker processes to execute transactions, which modify their own versions of memory safely by for example relying on copy-on-write support in the OS [12]. Transactions then rely on the TM system both to access the correct version of memory given some VID, and to atomically commit all speculative writes of a transaction, even if they come from different threads given the VID.

In these systems, explicit communication is required both for passing speculatively memory modifications between pipeline stages (uncommitted value forwarding), and for sending records of speculative memory accesses to the commit process (speculation validation). This is because these systems all use a "privatized by default" memory model via process separation. Because all memory is privatized through forking except for that which is explicitly `mmap`'d to shared memory, any accesses that need speculative verification must be explicitly communicated to the commit process via software queues.

Additionally, all speculative memory modifications and their addresses that may be needed by later pipeline stages must be explicitly sent via software queues to later pipeline stages working on the same transaction, in addition to the commit process. This must be done both for uncommitted value forwarding and for speculation validation.

The amount of this required communication and the resulting performance impact it has is dependent upon the complexity of the program being parallelized and the abilities of the method of parallelization (e.g. automatic vs. manual). For example, in prior MTX STM systems [12, 13], expert programmers performing laborious manual pipeline parallelization ensured that speculation validation was low via minimal read and write sets in order to achieve speedup on complex programs.

Figure 2.2 demonstrates how heavily performance depends upon read and write sets sizes when using these systems. Whole program speedup is compared for two versions

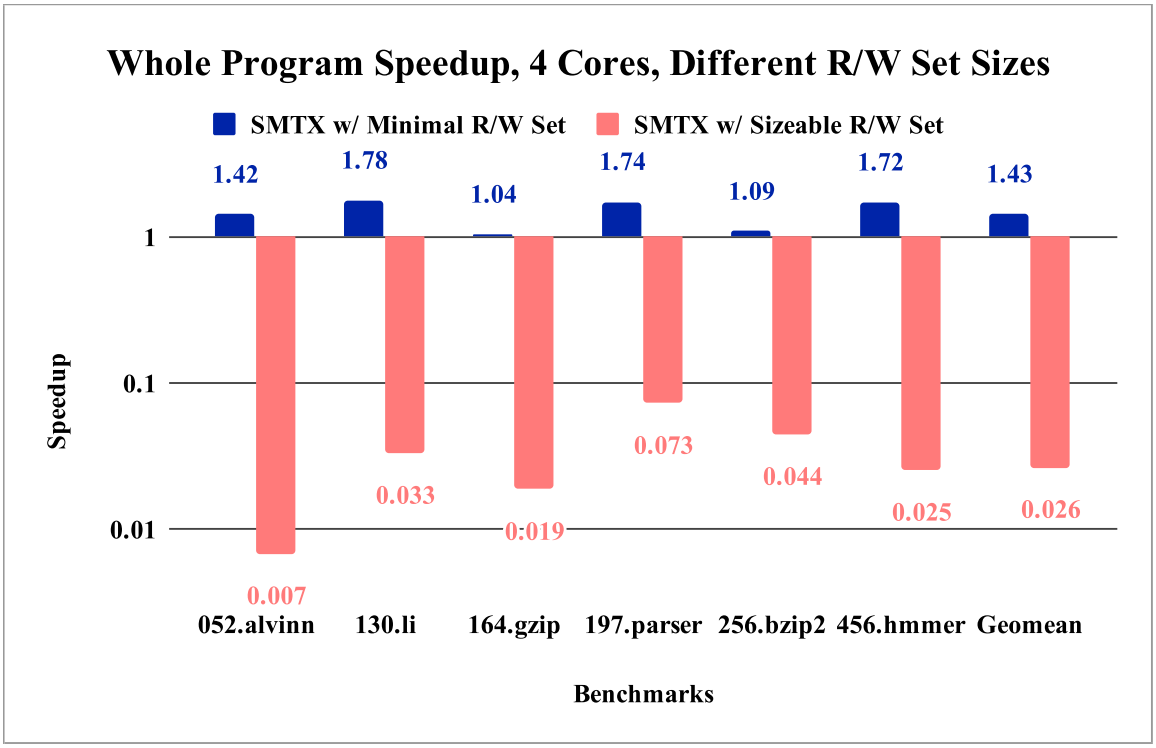


Figure 2.2: On an MTX STM system, whole program speedup over sequential execution with a minimal R/W set vs. a substantial R/W set.

of programs using an MTX STM system: one with a minimal read and write set that was manually transformed by an expert, and one with speculation validation added to shared data accesses to approximate what a compiler might require with basic static analysis. As expected, more speculation validation turns slight speedups into substantial slowdowns.

Similar results have been presented in the context of automatic speculative parallelization. For example, one past work [29] used an MTX STM system to automatically pipeline parallelize a script interpreter specialized with simple scripts. Performance was again limited by the overheads of software speculation validation; significant effort was spent on optimizing speculation validation communication. Speedup was achieved by scaling up the number of threads to overcome these overheads, making the technique not well suited for consumer grade systems.

Similarly, Johnson [20] demonstrated the need for low overhead speculation validation in the context of automatic parallelization. He found that for simple benchmarks executing

in a DOALL fashion using an STM system [30, 31], "imprecise analysis forces the compiler to compensate with more speculation . . . Increased validation overheads cause application slowdown." More complex programs (such as those from the SPEC benchmark suite [32]) require parallelization models such as DOACROSS or DSWP, and many have yet to be profitably automatically parallelized.

Thus, even with the strongest modern static analyses, automatic parallelization often requires speculation with sizeable read and write sets. The resulting validation overheads can make it difficult for these parallelized programs to achieve speedup. Instead of hoping for future heroic static analyses, or relying on significant expert programmer effort for manual parallelization, this dissertation embraces an alternative approach: make speculation validation cheap in order to make speculative parallelization less laborious and more feasible.

Chapter 3

Design Overview of the HMTX System

By providing support for low-overhead, resilient multi-threaded transactions, HMTX enables the profitable parallelization of complex programs with substantial speculation validation. This section provides an overview of the first HMTX design that overcomes the challenges required to execute complex parallelized programs in modern systems.

3.1 Overview

Similar to past MTX proposals, the HMTX system allows for different versions of memory, where multiple versions of a single address can exist simultaneously. Every transaction is assigned a *version ID* (*VID*), and all memory operations inside each transaction are labeled with this VID.

These VIDs are unsigned integers of some specific bit width (see Chapter 5.3.2 for more details). Note that VID $= 0$ represents non-speculative state; for example, normal, non-speculative loads and stores use VID $= 0$.

VIDs correspond to the original, sequential program order of the transactions; given a speculative store with VID x, a speculative load with VID $< x$ should not see that speculative store, while a speculative load with VID $\geq x$ should see it. If an address is read by a transaction with VID $y > x$ and then a speculative write occurs to that address with VID x,

all transactions with VID $\geq y$ should abort due to a read-after-write data hazard violation. To facilitate this, speculative memory accesses from an HMTX mark memory in the caches they touch with their VID, allowing for two key properties which are requirements of an MTX system:

1. **Group transaction commit**: The speculative modifications from many distinct threads working on the same transaction should be atomically committed as a group. These threads are likely on different cores; hence, atomic commit must be provided for all speculatively accessed memory from this transaction across multiple cores and caches.
2. **Uncommitted value forwarding**: An uncommitted memory modification from one pipeline stage of a transaction should be seen by later pipeline stages working on the same transaction. Additionally, uncommitted values from a transaction should be visible to later transactions according to the original sequential execution order of the program. Many works provide uncommitted value forwarding as an optimization [33, 15, 28, 34, 35, 36], however for MTXs it is a requirement.

To provide support for group transaction commit, VIDs that need to be committed are sent to the memory system, and then all lines with these VIDs can be committed together. To provide support for uncommitted value forwarding, speculatively modified memory marked with VID x can be seen by accesses marked with VID $y \geq x$. Program correctness will be maintained because these VIDs correspond to original sequential program order (Chapter 7).

3.2 New HMTX Instructions

New instructions must be added to the instruction set architecture (ISA) in order to support MTX, listed in Table 3.1.

First, transactions must signal when they begin and end. End does not mean commit;

Instruction	Description
`MTX_BEGIN(VID)`	Enter or exit a transaction. Used to move between different MTXs or back to non-speculative execution (VID equal to zero). Sets the VID register in the core to `VID`.
`MTX_COMMIT()`	Atomically group commit a transaction with VID of the current active transaction in the VID register set by `MTX_BEGIN()`.
`MTX_ABORT(VID)`	Abort all transactions active in the system. All threads should be interrupted and redirected to the handler function they registered with `MTX_INIT(handlerFunction)`.
`MTX_INIT(handlerFunction)`	Before speculative execution begins, set the location of a recovery handler function that must be run in the event of an abort. Should be called by all speculative worker threads.

Table 3.1: New instructions added to the ISA. Note that all of these instructions include an implicit memory barrier.

the current pipeline stage may simply be done with its work on its part of the transaction, and can begin work on the next transaction, or work on another task entirely as long as it is not dependent upon its previous speculative tasks.

This proposal uses a single instruction to accomplish this: `MTX_BEGIN(VID)`. This instruction can be called to move between different MTXs or back to non-speculative execution (VID equal to zero). When called, it sets a newly added VID register in the core, specific to the thread context, to the provided VID. This VID is attached to all memory operations in the system that follow `MTX_BEGIN` in program order.

Next, a `MTX_COMMIT(VID)` instruction is added in order to signify that a particular MTX should atomically group commit. Commit must only be called once by one of the threads participating in the transaction, and only when no more speculative accesses will be made using this VID. In DSWP, this is once the final pipeline stage has finished. Commits are must be executed in order, as coordinated by software.

An `MTX_ABORT(VID)`[1] instruction is also added in order to explicitly signal an abort

[1]This is similar to `resteer` as discussed by Vachharajani [1]. This allows for one thread to preempt execution of other threads that are participating in speculative parallel execution during abort, and "resteer" them to thread-local recovery code.

due to some misspeculation condition detected by the software, such as control flow misspeculation. During an abort, either triggered explicitly or implicitly via a misspeculated load or store, there must be some recovery code for the threads to jump to in order to take some action and continue execution. Therefore an `MTX_INIT(handlerFunction)` instruction must be used to set the location of this recovery code prior to speculative execution beginning on every thread.

Note that all of these instructions include an implicit memory barrier. Without this, complications could arise in the design of the HMTX system, as the coherence protocol design is intended to handle and track all dependences between different VIDs which represent different transactions. Otherwise, there could be memory operations reordered with incorrect VIDs, or forwarding between memory operations from different transactions that are not seen by the protocol. These instructions are relatively rare and so this does not have a meaningful performance impact.

3.3 MTX Instruction Usage

Figure 3.1 shows a code example comparing the sequential, DOACROSS with TLS, and speculative DSWP with MTX versions of a simple linked list traversal program, where the DSWP version uses the instructions from Chapter 3.2.

First, note that in the sequential version (Figure 3.1(a)), the early exit check `if (w > MAX)` limits parallelization. If this control dependence is not somehow bypassed or deferred then profitable parallel execution is very difficult to achieve, as the next parallel iteration cannot begin until the majority of the current iteration's work is complete. Instead, DOACROSS and DSWP control flow speculate this dependence does not exist. In DOACROSS it is not checked until at the end of each loop. In DSWP is not checked until stage 2, after later iterations (in original program order) have already begun in parallel. If this speculation is incorrect in either case then all later transactions are aborted.

(a) Original Non-Speculative Sequential Program

```
while (node):
  w = work(node); // May modify order of list
  if (w > MAX): break;
  node = node->next;
```

(b) DOACROSS with TLS

```
// Assume initial node already produced
while ((node = consumeNode()) != nullptr):
  beginTX();
  produceNode(node->next);
  w = work(node->data); // May modify order of list
  commitTX();
  if (w > MAX): abortRestOfTXs();
```

(c) Speculative DSWP Stage 1 Parallel Version, Using HMTX

```
MTX_INIT(abortHandlerFunction);
register vid = 0; // Stored in register
register leaveLoop = (node == nullptr); // Stored in register
while (!leaveLoop):
  vid += 1;
  MTX_BEGIN(vid);
  // Creates new version of node w/ VID=vid:
  node = node->next; // Note:  node is global/visible to all threads
  leaveLoop = (node == nullptr);
  MTX_BEGIN(0); // Does not commit
  produceVID(vid);
produceVID(0); // Signal end of the loop
```

(d) Speculative DSWP Stage 2 Parallel Version, Using HMTX

```
MTX_INIT(abortHandlerFunction)
while (vid = consumeVID()):
  MTX_BEGIN(vid); // Continue TX started in Stage 1
  // Finds correct node version w/ VID=vid:
  w = work(node); // May modify order of list
  // Commit the current active vid from the most recent MTX_BEGIN(vid):
  MTX_COMMIT();
  if (w > MAX): MTX_ABORT(); // Aborts all uncommitted transactions
```

Figure 3.1: Pseudocode of execution paradigm options for an example program, with sequential (a), DOACROSS with TLS (b), and speculative DSWP with HMTX (c, d) versions. In DSWP, Stage 1 (c) does not commit; after each iteration, it produces the VID of the transaction it just completed for Stage 2 (d) to consume and continue with execution. On abort, both stages execute their handler functions (set prior by `MTX_INIT`), and the queues would be flushed.

Transactions are first started by the initial pipeline stage (Figure 3.1(c)) via `MTX_BEGIN(VID)`. All memory operations from this thread from this point forward come from this transaction, and hence any memory read or written will be marked as such. Once stage 1 has completed its portion of the transaction, it calls `MTX_BEGIN(0)`, which represents that the program is moving back to non-speculative execution, but is **not** committing. All work done between `MTX_BEGIN(0)` and `MTX_BEGIN(vid)` at the top of the loop is essentially bookkeeping, e.g. producing to stage 2 (via a software queue) the VID of the transaction it just finished its portion of work on, and checking that the loop should continue iterating.

Next, note that instead of requiring explicit queue operations for communicating dependences between stages, HMTX's versioned memory can be leveraged. For example, `node` can be communicated by stage 1 to stage 2 via a single speculative store to the shared global `producedNode`, and stage 2 can load it via a single speculative load to `producedNode`. All speculative modifications are marked with the VID of each transaction, meaning each transaction's version of `producedNode` exists in memory, identified by their VID. These versions are accessible by other threads if they are using the same VID (Chapter 4.1).

Figure 3.1(d) shows stage 2, where transactions can continue with execution and eventually commit. A stage 2 thread continues with execution of some transaction that was previously started by stage 1 with VID (determined via a consume from a software queue), entering that transaction via `MTX_BEGIN(VID)`, just as stage 1 did. All memory operations are again marked with the VID of the transaction, meaning that any memory modifications done by the prior thread inside the same transaction are visible to this thread, even though they were performed by a different thread and remain uncommitted.

Additionally, note that stage 1 speculatively accesses `node->next`. This is done with some VID $= x$. If at some later time a transaction in stage 2 with VID $y < x$ attempts to modify `node->next` (e.g. during the `work` function), then an abort is triggered due to a read-after-write violation (Chapter 7).

Finally, stage 2 completes its part of the transaction and then commits it entirely, including all modifications from the stage 1 thread. This is done via `MTX_COMMIT(VID)`, which commits that specific VID and returns to non-speculative execution.

While this code works as a two thread pipeline, it could also be executed via PS-DSWP, with multiple threads executing stage 2. Any data dependence issues between concurrent calls to the `work` function would again be detected thanks to the HMTX system, and an abort would be triggered (Chapter 7).

3.4 Supporting Complex, Long-Running Transactions

Prior HTM systems [14, 15, 16, 17, 18] do not provide sufficient support for long-running and complex transactions that are often required for long-running and complex programs. Chapter 5 discusses solutions to the following problems that would otherwise inhibit performance or make parallelization impossible:

- **False misspeculation due to branch misprediction (Chapter 5.1):** In processors with large pipelines and branch prediction, loads are often speculatively executed based on branch prediction. If a branch is mispredicted, any corresponding squashed loads that were dependent upon this branch that have already executed have no impact other than moving data around in the caches. However in the HMTX system, VIDs are marked on memory that is HMTX-based speculatively read. This can result in spurious misspeculations if not resolved, as memory is incorrectly marked as speculatively accessed by VIDs due only to branch misprediction. To our knowledge, no past work has recognized or solved this issue, likely because most past systems either used relatively small transactions, parallelized programs without complex control flow that resulted in significant branch misprediction, or both.

  HMTX solves this by introducing a new structure similar in complexity to a store queue to track HMTX-speculative loads that are executed branch-speculatively.

- **Aborts caused by processor interrupts (Chapter 5.2):** Given long-running transactions with complex memory access patterns, interrupts and exceptions are commonplace, e.g. due to preemption or virtual memory management. These operations must not cause misspeculation.

  HMTX solves this by differentiating between instructions that are part of the original program and those that come from the operating system that do not impact correctness of the HMTX-speculatively executing program.

- **Long-running and expensive commit processing (Chapter 5.3):** A naïve system may need to keep track of all speculatively accessed memory in order to explicitly transition them on commit. This would require some structure in hardware or software to scale along with the amount of accessed memory (which is quite large in the evaluated benchmarks (Chapter 8.4)), increasing complexity and degrading performance of the system in execution time and energy.

  HMTX solves this through a simple commit scheme that allows for speculative cache lines to lazily transition to non-speculative state.

- **Aborts caused by large read sets overflowing cache (Chapter 5.4):** Speculatively modified values must have their original non-speculative counterparts backed up until commit. These backups can increase cache pressure, and potentially force an abort if all speculatively modified memory cannot fit inside the caches. Most prior systems that allow for overflow of speculatively accessed memory outside of caches require bookkeeping to track this speculative memory, which increases complexity and/or degrades performance.

  HMTX solves this by the design of its protocol, allowing for some speculatively read lines to be evicted from the last level cache.

Chapter 4

Detailed Design of the HMTX System

This Chapter details the design of the HMTX system, including the cache coherence protocol extensions (Chapter 4.1) to support multiple versions of cache lines (Chapter 4.2); detailed descriptions of speculative states and their implementation (Chapter 4.3); how requests and lines operate in this system (Chapter 4.4); how commits and aborts are handled (Chapter 4.5); efficient comparisons for VIDs in the cache system (Chapter 4.6); handling moving between versions (Chapter 4.7); handling outstanding requests from caches in the HMTX system (Chapter 4.8); what happens when different speculative lines overflow caches (Chapter 4.9); privatizing different versions of memory (Chapter 4.10); and operating system and program support required for software using HMTX (Chapter 4.11).

4.1 Cache Coherence Protocol Modifications

The base design of the system uses a snoopy MOESI cache coherence protocol [37]. In MOESI, there are 5 states: Modified (M), Owned (O), Exclusive (E), Shared (S), and Invalid (I). Modified and Exclusive lines are writable, meaning there are no other copies in the cache system and therefore writes can proceed unhindered. Owned and Shared lines are read only, allowing for the sharing of data across many caches. If a line in Owned or Shared needs to be written then an upgrade must be issued, invalidating other copies in the

cache system. Lastly, Modified and Owned lines are dirty, and must eventually be written back to memory. Shared and Exclusive lines are clean and can be silently invalidated and replaced if necessary.

The behavior of the MOESI protocol is unchanged when all lines and requests are non-speculative. VID $= 0$ represents non-speculative requests and lines, while VID > 0 represents speculative. When a line is speculatively accessed, it will transition to one of the newly introduced "speculative" coherent states: Speculative-Modified (`S-M`), Speculative-Owned (`S-O`), Speculative-Exclusive (`S-E`), and Speculative-Shared (`S-S`). This is similar to others [15].

In HMTX, differentiating between a Speculative coherent state versus its base non-speculative MOESI analog is accomplished via a bit called the "Speculative Bit" (SB). This bit is added to every line, alongside the other bits used for tracking MOESI States. For example, a line in the Modified MOESI state is the same as a line in the Speculative-Modified state, except it also has its SB asserted as well, while the non-speculative Modified state does not.

If SB is set to true (and thus the line is in a speculative coherent state), then hits and misses behave differently per these speculative coherent states. This is explained further in Chapter 4.4.

4.2 Cache Line Versioning

Multiple versions of the same cache line can exist in a single cache set. In order to differentiate these versions, each line has two VIDs added to it: the *modifier VID* (modVID), and the *highest accessor VID* (highVID). The modVID corresponds to the VID of the transaction that created this version of the line due to a speculative modification. Meanwhile, the highVID corresponds to the highest VID which accessed this version of the line.

The notation `S-M(m,h)` means that an S-M line has modVID `m` and highVID `h`. The

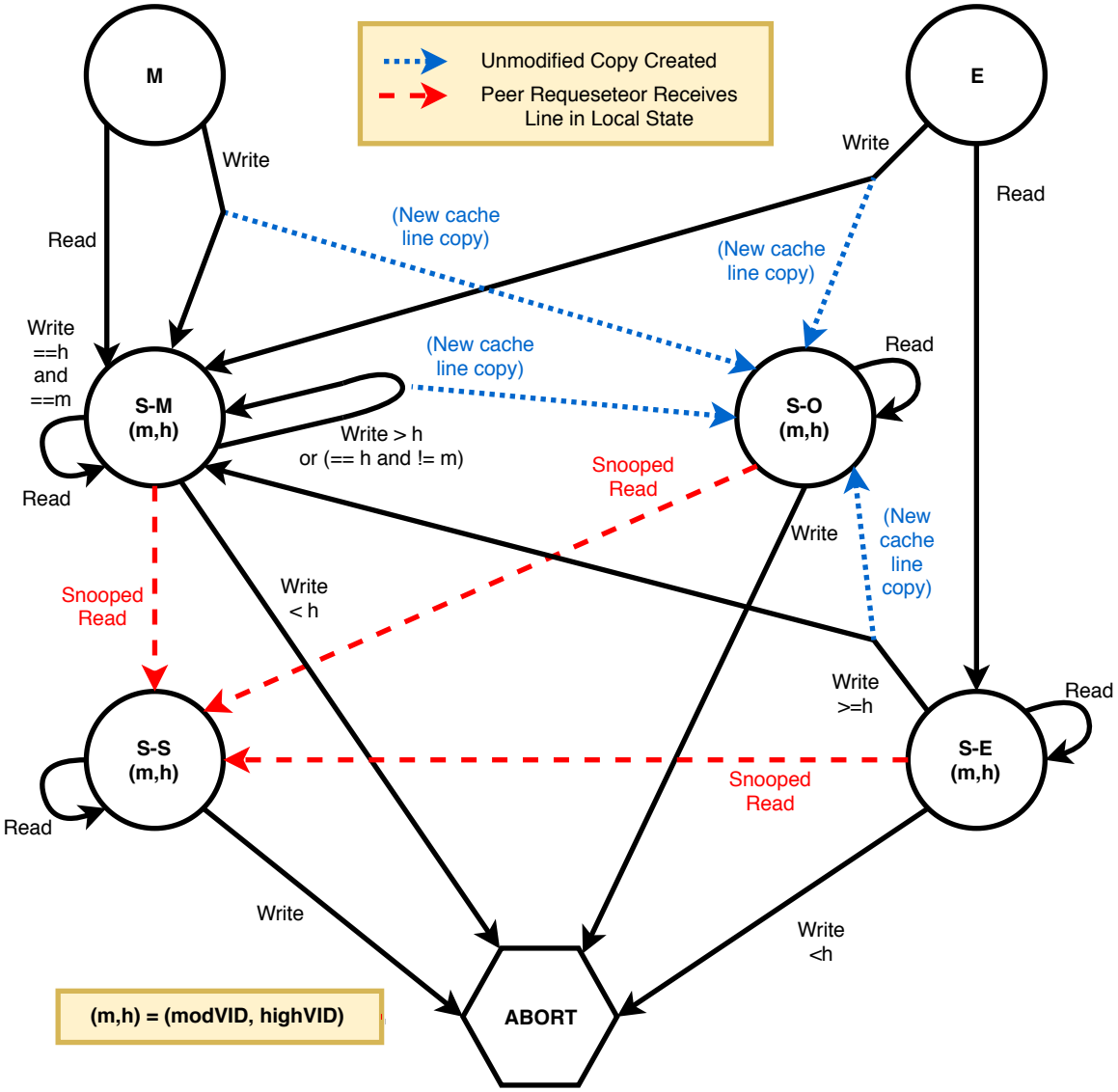


Figure 4.1: State diagram for speculative accesses. All accesses are assumed to be speculative, i.e. with non-zero VID. If a transition label does not mention a condition check for an access (e.g. "Write < h"), then it is assumed the access hit the version of the line given the conditions mentioned in Chapter 4.3. O, S, and I states are not shown for simplicity; they would follow the same path as M or E once acquiring exclusive access.

VIDs are always listed in this order when seen in this tuple notation.

Non-speculative lines continue to determine hits and misses based on tag comparison and the line's coherent state. For speculative lines, an additional check is required; hits and misses are dependent upon comparing the VID of the request to modVID and highVID. Note that the request's cache set index is still dependent only on the request's address.

If a cache set fills up, then any of the versions can be written back to the next level cache as normal. The writeback must include the modVID and highVID along with the address and data of the line. Note that selecting some speculative versions of cache lines as a victim for writeback past the last level cache supporting MTXs forces an abort (Chapter 4.9).

As noted above, the modVID corresponds to the VID of the transaction that created this version of the line due to a speculative modification. As previously mentioned, all non-speculative versions of a line have a modVID of zero (for example, a line in S-E with modVID $= 0$ represents a non-speculative line that has been speculatively accessed). A speculative modification creates a new cache line in state S-M and sets its modVID to the VID of the speculative store. An unmodified copy of the line is kept in S-O, retaining the modVID of the original line and its original data.

This can be seen in Figure 4.1, with any speculative modifications that do not trigger an abort going to state S-M, and also creating an unmodified copy in S-O. Using these lines, reads with lower VIDs can find their correct version of the line, avoiding write-after-read hazards.

As mentioned, the highVID corresponds to the highest VID which accessed this version of the line. A naïve scheme might only keep a single VID per line to track what version it came from. However, this means that if a transaction only reads a line then we need to create a new copy of the line with the same data but different modVID. Instead, high-VIDs allow for tracking reads without creating duplicate lines. This allows for supporting uncommitted value forwarding more efficiently and reduces cache pressure.

The use of these two VIDs allows us to represent multiple conceptual cache line versions with a single physical line, assuming those versions all share the same data. Additionally, it allows a speculative modification to check a single cache line to determine if an abort must be triggered due to a dependence violation. Finally, it is used to determine which particular lines should hit given the accesses' VID, and for simplifying discarding versions of a line that are no longer needed during commit.

4.3 Description and Operation of Speculative Lines and Requests

Qualitatively, the states can be thought of as follows:

Speculatively Modified (S-M) lines represent the "latest" dirty speculative version of the line, meaning this version of the line is the latest with respect to original program order, and so no version of the line exists with a higher modVID. The highest VID to access this line is set to highVID. Thus, if the VID of a speculative write is greater than or equal to highVID, it can proceed without triggering an abort, as no "later" access has already occurred to the line.

Speculatively-Owned (S-O) lines represent lines that were previously speculatively modified but can no longer be written without triggering an abort due to a potential dependence violation, as some "later" access already occurred to the line. Specifically, some other speculative write must have occurred with VID $==$ highVID, and that speculative write created a new copy of the line in S-M line with its speculative modifications, leaving this line in S-O behind unmodified.

Speculatively-Exclusive (S-E) lines are essentially the same as S-M, except no modifications to the line have occurred, whether speculatively or non-speculatively. Consequently, on commit the line can return to a clean non-speculative state (Exclusive or Shared) instead of a dirty non-speculative state (Modified or Owned), preventing unnecessary write-back to memory. This state can never have modVID > 0.

Speculatively-Shared (S-S) lines are used to allow for shared copies of speculatively accessed lines to exist in different caches. This enables efficient sharing of read-only speculative accessed data, which is important for many TLP programs. This version of the line does not respond to snoops, as one of the S-M, S-O, or S-E versions will respond instead.

Table 4.1 shows some properties of these states, and Table 4.2 qualitatively describes each of these states in further detail.

Line Coherent State	Speculatively Modified?	"Latest" Version?
S-M	modVID == 0 ? N : Y	Y
S-O	modVID == 0 ? N : Y	N
S-E	N (modVID must equal 0)	Y
S-S	modVID == 0 ? N : Y	N

Table 4.1: Properties of the speculative coherent states.

Hits and misses are determined by combining the address and coherent state of the line (including the Store Bit (SB)) as in traditional coherent cache systems with these VIDs of the line and the VID of the received request. Given some speculative line, for an incoming request with VID a:

- **S-M/S-E(m,h):** `if` $(a \geq m) \Rightarrow$ `hit`
- **S-O/S-S(m,h):** `if` $(a \geq m)$ `and` $(a < h) \Rightarrow$ `hit`

The basic additions necessary for implementing this logic can be seen in Figure 4.2.

If the line has $m == 0$ (i.e. it is the non-speculative version of the line) then $(a \geq m)$ must be true, and so non-speculative requests (with $a == 0$) will always hit S-M/S-E lines. Additionally, in the base protocol, it is assumed that $h > 0$[1], which means non-speculative requests will always hit S-O/S-S lines as well. This intuitively makes sense, as if the line has $m == 0$ then this is a non-speculative version of the line that was speculatively accessed. Thus, all non-speculative versions of lines will be hit by all non-speculative requests.

These are the same conditions used to determine hits for snooped requests on the bus. However, as noted, S-S lines ignore snooped requests, similar to the S state in the normal MOESI protocol.

The modVID and highVID essentially act as a minimum-maximum range of VIDs, used to determine what accesses hit what lines, and when to trigger misspeculation. The protocol is designed such that an incoming request to a cache knows if it should hit, miss, or trigger misspeculation solely by using the coherent state of each line, their modVID and highVID, and the VID of the request. For example, there is no "potential" hit case, wherein global

[1]This assumption is relaxed for efficient commit process handling, discussed further in Chapter 5.3.

State	What is m?	What is h?	Description
S-M(m,h)	m represents the transaction which created and modified this line.	h represents the "latest" transaction which read this line.	Speculative-Modified: A dirty version of a cache line. Transactions with VIDs $\geq m$ may have speculatively read this line, with the greatest of these VIDs being h. Note that m can be $== 0$, representing a dirty non-speculative line was read by one or more transactions. Otherwise $m > 0$, and so this version was created and modified by one or more writes from transaction with VID m. If $m > 0$, then there must exist another version of the line in the cache system in S-O(m', h'), with $h' = m$, containing the previous data in the line prior to speculative modification by transaction with VID m. There may exist other speculatively modified versions of the line in S-O, as well as S-S lines to allow for efficient sharing of data. No S-E version of the line exists.
S-O(m,h)	m represents the transaction which created and modified this line.	h represents the "latest" transaction which tried to write to this line after m.	Speculative-Owned: Some version of a line which had a "later" transaction (i.e. VID $> m$) attempt to write to it. Thus, this line was left as a copy from transaction VID m, but without the modifications by VID h. Note that m can be equal to 0, representing some non-spec version of a line was speculatively modified by the transaction with VID h. For example, if the line existed in M (i.e. non-speculative modified) and then was written to by a transaction with VID $= 2$, then we would end up with two lines: one in S-M(2,2) with the speculative modification, and one in S-O(0,2) without the speculative modification. S-O lines are never a version of the "latest" transaction, and as such any writes that hit such a line signify that an abort should occur. Note that m cannot equal h, because S-O lines can only be created when some S-M or S-E line had a write from transaction with VID $> m$, or some M or E line (i.e. VID 0) had any speculative write (i.e. VID > 0). No S-E version of the line can exist anywhere in the cache if this version exists.
S-E(m,h)	m must equal 0, i.e. the non-speculative VID.	h represents the "latest" transaction which read this line.	Speculative-Exclusive: The non-speculative version of the cache line, which has been speculatively read by largest VID h. Only speculative reads have occurred to addresses in this line; no speculative writes have been made to addresses in this line by any transaction. No other version of the line can exist in the cache system (except potentially S-S for efficient sharing of read-only data).
S-S(m,h)	m represents the transaction which created and modified this line.	h represents the "latest" transaction which may have written to this line after m.	Speculative-Shared: A read-only copy of another speculative line. There may be other S-S versions in different caches. There must be an S-M, S-E, or S-O version of the line in another cache. This version of the line is simply used to facilitate performant sharing across caches. It does not respond to snoops. It can be silently evicted.

Table 4.2: Description of speculative states. (m,h) represents the modVID and highVID. Note that VIDs correspond to the execution order of the original sequential program. Therefore, speculative modifications to lines that have already been read or written by a larger VID signal a potential data hazard and misspeculation.

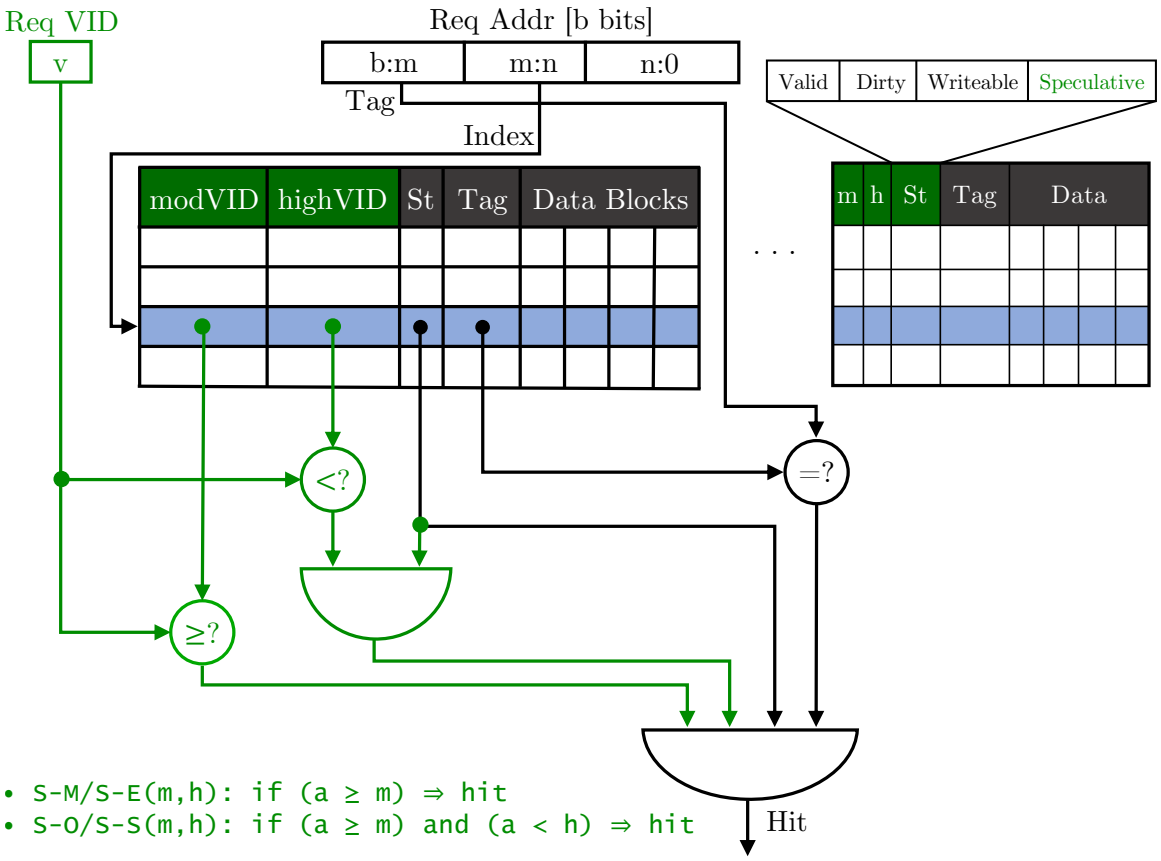


Figure 4.2: Circuit diagram displaying the basic logic needed to implement new hit logic required for the HMTX speculative coherence protocol. "St" represents the status bits per cache line, i.e. Valid, Writable, Dirty, and the newly added Speculative. Items in green are added for HMTX.

knowledge of all versions of this line must be gathered before determining if a request in the local cache should have hit or missed this line. Note that this is an important property of any cache coherence system in order to be performant, but it is non-trivial to maintain this property when implementing a cache system with multiple versions spread across the cache hierarchy.

Global knowledge of the state of every version of the line is either implicit or not required to determine the correct course of action. Requests will only hit on one version of the line. If that version is not present, a request is broadcast on the bus with the request's VID. Only one cache will respond, with the line that should have hit for this request had it been in the same cache.

Finally, lines do not need to know the state of other lines in other caches to determine what state to transition to in the case of a commit or abort (Chapter 4.5). This enables more efficient commit and abort processes. Similar to before, this is a property that other HTM systems often have as well, but it is harder to maintain when dealing with multiple versions of lines spread across the cache hierarchy.

The following is a list of invariants that always hold for the states and VIDs of the lines, even when they are changed due to a new read, write, commit, or abort:

- There can only be one S-M version of the line at a time;
- If there is an S-O version of the line then there must exist an S-M version of the line;
- If there is an S-E version of the line then there may not be any other versions of the line except in S-S;
- The S-E version of the line represents the latest committed, non-speculative version of the line (with modVID $== 0$);
- The line with the greatest highVID of any lines can only ever be in S-M or S-E;
- For states S-M, S-E, and S-O, there cannot be another line in one of these states with the same modVID;
- If there exists an S-S line, there must be some other S-M, S-E, or S-O line with the same modVID, and with $\geq$ highVID.

Table 4.3 summarizes the allowed states for other versions of the line in the cache (columns) given some version with a particular state exists (rows), displaying some of these invariants.

4.4 Operation of Speculative Accesses

When a line is first speculatively accessed, writable (M or E) access must be gained for the line in the L1 cache, as seen in Figure 4.1. This means if the line is not in the cache then a read exclusive request is sent out on the bus. If the line is in the L1 but is not writable

	M	O	E	S	I	S-M	S-O	S-E	S-S
M	✗	✗	✗	✗	✓	✗	✗	✗	✗
O	✗	✗	✗	✓	✓	✗	✗	✗	✗
E	✗	✗	✗	✗	✓	✗	✗	✗	✗
S	✗	✓	✗	✓	✓	✗	✗	✗	✗
I	✓	✓	✓	✓	✓	✓	✓	✓	✓
S-M	✗	✗	✗	✗	✓	✗	✓	✗	✓
S-O	✗	✗	✗	✗	✓	✓✓	✓	✗	✓
S-E	✗	✗	✗	✗	✓	✗	✗	✗	✓
S-S	✗	✗	✗	✗	✓	✓	✓	✓	✓

Table 4.3: Permitted cache states of a cache line. Note that the MOESI portion of the table (upper left quadrant) is unchanged from the default MOESI implementation. Each row represents the state for a line that exists in the cache system. The column in each row represents if another line with the same address and different VID must exist (✓✓), may exist (✓), or cannot exist (✗) in the cache system. For example, if an S-O line exists, then in the cache system there must be an S-M version (✓✓), and there may be another S-O version or S-S copy (✓). Note that a line cannot exist in both a speculative and non-speculative state simultaneously, as seen in the the lower left and upper right quadrants. That is, if a speculative line exists, then no non-speculative valid line (MOES) can also exist, and vice versa.

(i.e. Shared or Owned), then an invalidation must be broadcast in order to gain exclusive access.

Once the cache has a writable copy, the request can proceed. In the case of a read, the line is moved to S-E (if the line was still clean (E)) or S-M (if the line was already dirty (M)) with the VID x of the request set on the line as the highVID. modVID is left as zero because this is a read, so the non-speculative version of the line feeds the read and no new version is created. Figure 4.3 shows an example of this case at instruction 1 with VID 1, resulting in state `S-E(0,1)`.

If a speculative write with VID $y \geq x$ is received, then a copy is made of the line to preserve the non-speculative state, which was the version x used. The resulting states would be `S-O(0,y)` and `S-M(y,y)`. Again, this is seen in Figure 4.3, with VID 1 at instruction 2, and corresponding new versions `S-O(0,1)` and `S-M(1,1)`.

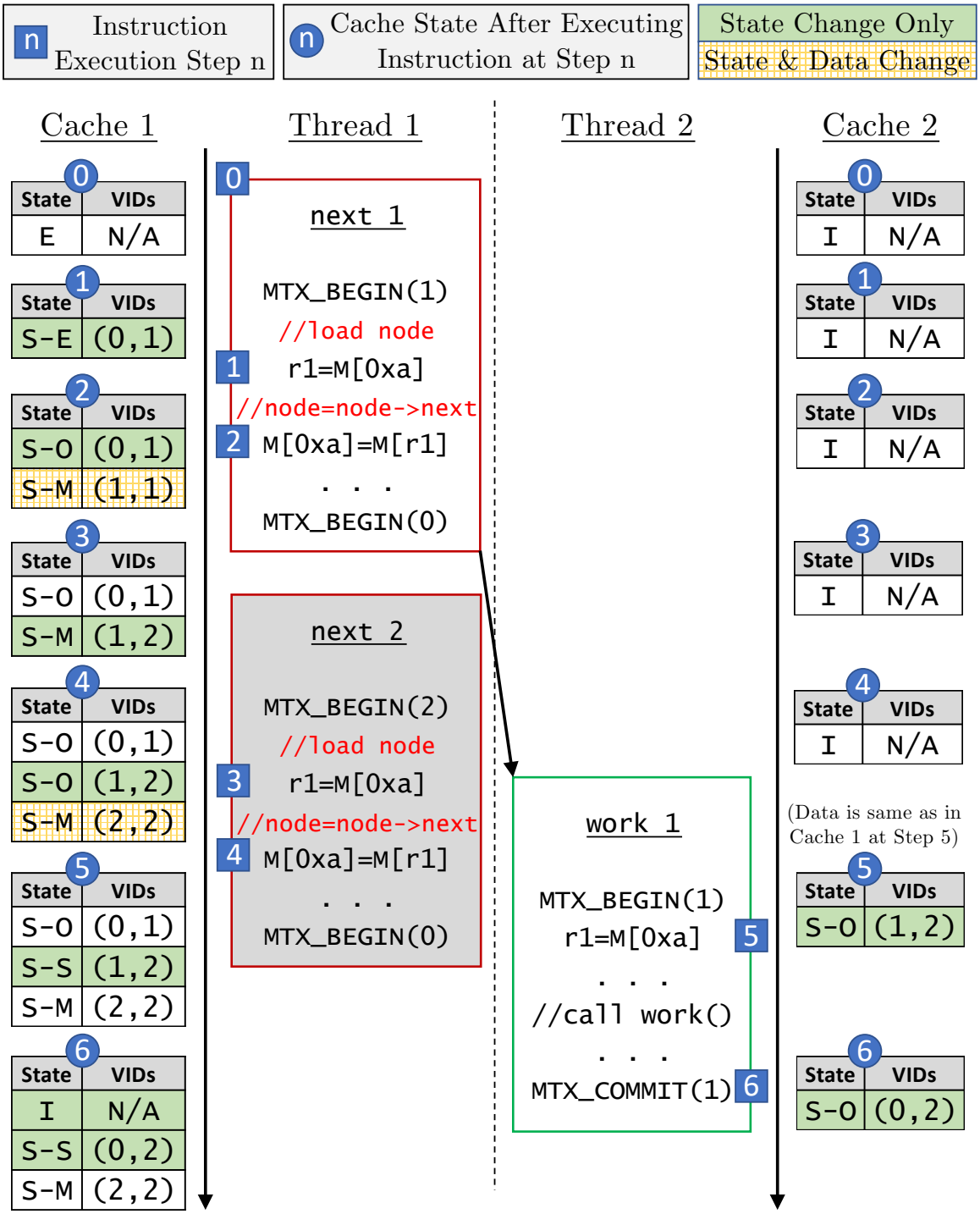


Figure 4.3: Pseudocode and cache states of the Figure 3.1 example. Step 0 signifies the state prior to entering the parallelized loop. Note that cache state is only shown for address `0xa`. All of these lines could exist anywhere in the cache. Lines with solid green backgrounds have had their coherent state and/or VIDs changed but not the data from the line itself. Lines with checkered yellow backgrounds have additionally had the data modified.

Continuing with the example in Figure 4.3, a speculative read occurs with VID 2 at instruction 3. In this case the `S-M(1,1)` version simply updates its highVID to 2, as it is now the highest VID to have accessed this line. Note that modVID stays the same as no new version was created of this line because this was not a write.

Next, a speculative write executed for VID 2 at instruction 4. In this case the `S-M(1,2)` version transitions to `S-O(1,2)`, keeping highVID at 2 and keeping the data the same. Additionally a new version of the line including the speculative modifications is created with modVID and highVID set to the VID of the write, `S-M(2,2)`. Three different versions of the line now exist with different modVIDs and data.

When a read with VID 1 is received at instruction 5, it is broadcast on the bus and hits the `S-O(1,2)` version of the line due to the scheme as described in Chapter 4.1. The response is sent in `S-O(1,2)`, as seen in Figure 4.1. If an access with VID greater than or equal to 2 was received it would hit the `S-M(2,2)` version. Because these VIDs correspond to original program order, this ensures correctness of execution, as reads with VID 1 should not see the speculative updates by transaction VID 2 but any accesses with VID greater than or equal to 2 should see the modifications by VID 2.

Lastly, Figure 4.3 shows the final state after Thread 2 commits. The commit process is explained further in Chapter 4.5.

Note that many speculative copies of lines are kept across the cache system, and most cannot overflow the caches without forcing an abort (discussed further in Chapter 5.4). This means there is greater cache pressure. This can negatively impact performance, depending on cache sizes and the access patterns of the parallelized program. This effect is evaluated and discussed in Chapter 8.5.

4.5 Implementing Commits and Aborts

To reason about how commits and aborts occur, assume for now that on a commit or abort every line in each cache is inspected and immediately transitioned to a new state if necessary depending on its VIDs and state. An optimized, lazy version is introduced in Chapter 5.3.

A commit or abort for some VID is processed via a broadcast on the shared L1-L2 bus along with the VID. The software must ensure that commits always occur consecutively (Chapter 4.11); otherwise behavior of the system is undefined.

On a commit for some VID x, lines transition as follows:

- **S-M/S-E**: If $x \geq$ highVID, the line moves to M if in S-M or E if in S-E. Else the line stays in its current coherent state, and if $x ==$ modVID then the line's modVID is set to zero. In either case, this version of the line is now the non-speculative version.
- **S-O/S-S**: If $x \geq$ highVID, this version of the line is invalidated. Else if $x ==$ modVID, the line's modVID is set to zero, as this version of the line is now the non-speculative version.

A state diagram for these commit transitions can be seen in Figure 4.4. Intuitively based on the design of the protocol, all lines with modVID $== x$ are now the committed non-speculative version, and hence they set their modVID $= 0$, as this represents non-speculative state. Additionally all lines with highVID $\leq x$ no longer need to be marked speculative at all, because all transactions that accessed them are complete. Therefore these lines can move to non-speculative coherent states (i.e. `S-M/S-E` $\rightarrow$ `M/E`, and `S-O/S-S` $\rightarrow$ `I`). Examples of such transitions can be seen in Figure 4.3, where a commit occurs at instruction 6, and cache lines transition accordingly.

On an abort for any VID, all uncommitted, speculatively modified memory in the cache system is flushed. This facilitates a simpler implementation; aborts should be very rare when a program is parallelized efficiently and thus the common case is optimized for, while slowdowns are pushed to the rare abort case. Aborts are further discussed in Chapter 4.9.

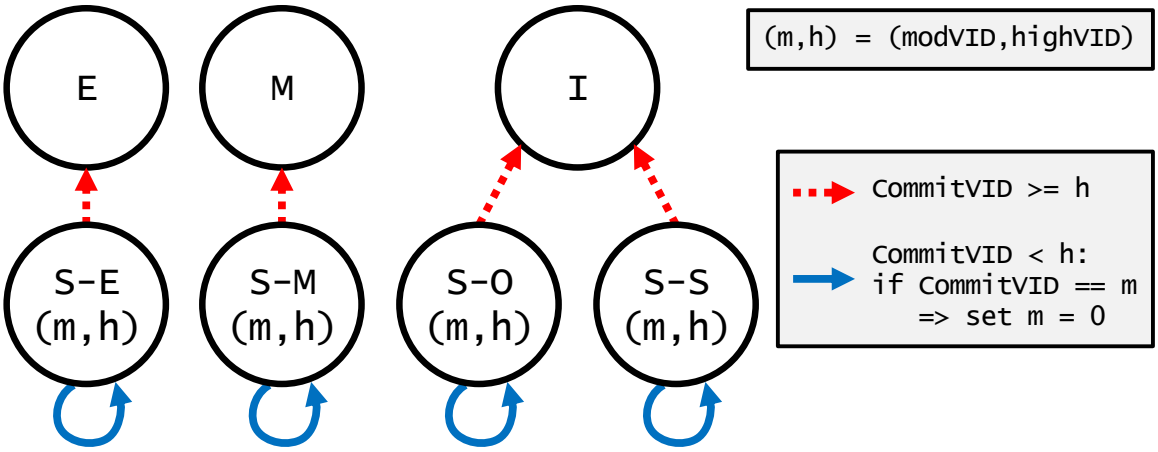


Figure 4.4: Commit state diagram.

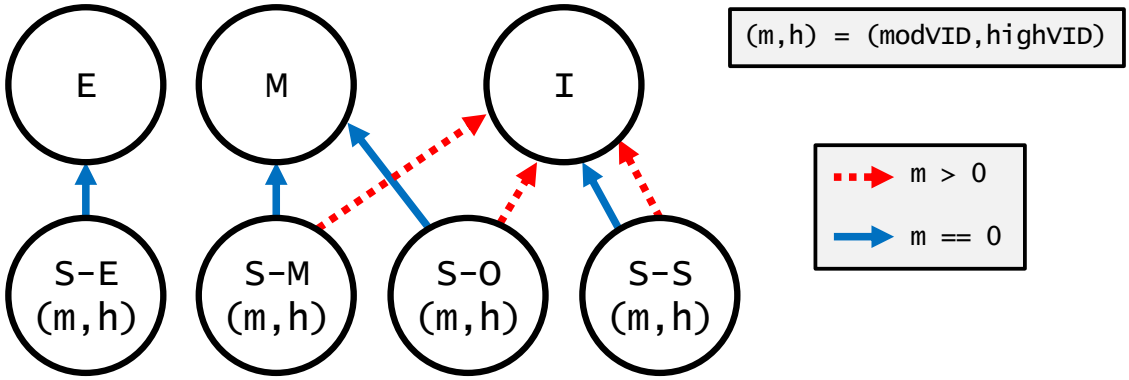


Figure 4.5: Abort state diagram. Note that S−E lines must have modVID $==$ 0, hence S−E has no transition for modVID $>$ 0. Also note that on an abort, all speculative memory in the cache is flushed, hence there is no concept of an "abort VID".

Intuitively based on the design of the protocol, lines that have modVID $==$ 0 are non-speculative, and therefore they should not be invalidated if they are dirty with respect to main memory. Otherwise, all other lines should be invalidated. This is reflected in the state diagram (Figure 4.5).

4.6 Efficient VID Comparisons

The majority of the area and power increases (Chapter 8.5.3) come from the two m-bit VIDs per line, along with comparing them to the incoming request VID when checking for

a hit.

In the evaluated implementation, $m = 6$. Instead of doing two full 6-bit comparisons on every cache set check, note that it is highly likely that VIDs in use by the system at any given time are equal or very close to each other. This is because each transaction has a single VID, and VIDs are used consecutively between transactions. Thus, a full 6-bit comparison is unnecessary for the large majority of accesses.

Instead, the highest 3-bits can check for equality while the low 3-bits can check for magnitude comparison. This keeps dynamic energy consumption low without compromising on the common case cache hit latency. In the very rare case that the low 3-bits are not equal, a cascading comparison can continue for the high 3-bits, delaying a cache hit while the comparison is completed.

Additionally, note that VIDs are unsigned integers. Recall that given some incoming request VID a, the comparisons against the per-line VIDs that are performed are $a \geq$ modVID and $a <$ highVID. If either of modVID or highVID $== 0$, representing non-speculative state, we can simply skip its comparison. That is, we know that that $a \geq 0$ will always be true, and $a < 0$ will always be false. Thus we can skip the comparison in these cases. This is common for much of the read set of a transaction, i.e. lines with modVID $= 0$ and in a speculative coherent state.

Lastly, when checking if an access hits a line in a non-speculative coherent state (i.e. lines with their Speculative bit set to false), no VID comparison is needed. These lines' VIDs are irrelevant; speculative and non-speculative accesses proceed based simply on tag comparison as usual and according to the normal MOESI hit logic.

4.7 Changing Between VIDs and Store-To-Load Forwarding

Care must be taken to ensure that the semantics of the HMTX system are sustained when moving between VIDs. The HMTX system relies on the coherence protocol design in the cache to ensure accesses to different versions of memory with different VIDs is well tracked. That is, it must ensure that loads and stores see the correct view of their versions of memory, and track what accesses hit what cache lines to ensure that aborts occur correctly.

Because of this reliance on the modified cache coherence protocol, store-to-load forwarding cannot forward between stores and loads from different VIDs (that is, between stores and loads from different transactions).

Store-to-load forwarding is a well known optimization technique wherein, given a store followed by a load to the same address, the store (yet to be committed and waiting in the store queue (SQ)) forwards its value to the load which is also uncommitted and waiting in load queue (LQ). Note that the load gets its value directly from the SQ in the core and so does not need to be sent to the cache.

An example is presented to illustrate why store-to-load forwarding cannot occur between stores and loads with different VIDs. Assume some memory location located at `0xa` has value `0` and exists on a line in non-speculative Modified state in some L1 cache. Then, we get the following series of instructions:

```
// Assume initially M[0xa] = 0
i1:  MTX_BEGIN(register1)      // Assume register1 = 1
i2:  store M[0xa] = register2  // Assume register2 = 2
i3:  MTX_BEGIN(register3)      // Assume register3 = 3
i4:  load register4 = M[0xa]   // register4 should now equal 2
```

First, assume that instructions execute and are committed one at a time, i.e. without store-to-load forwarding. Once instruction `i1` completes, the VID of all subsequent in-

structions should use VID 1 until a new `MTX_BEGIN()` instruction executes. Thus when `i2` completes, it uses VID 1 when sent to the cache. Once it is complete, there would be two versions in the cache of the line containing `0xa`: one in `S-O(0,1)` that contains the original data `0`, and another in `S-M(1,1)` that contains the new data `2`. Next, `i3` would execute, changing the VID of subsequent instructions to 3. Finally, `i4` would execute, which would hit `S-M(1,1)` and move it to `S-M(1,3)`, and set `register4 = 2`.

Instead, now assume that we have a LQ and SQ, and multiple instructions can exist in the pipeline at once, with stores forwarding to loads regardless of their VIDs. When `i2` executes, it now sits in the SQ waiting to be committed and sent to the core's L1 cache. Now assume `i3` changes the VID register to 3, and then we get to `i4`. It will sit in the LQ and `i2` would forward its value to `i4`. Now assume that `i2` is committed and finally sent to the cache. Again as before, there would be two version of the line containing `0xa`: one in `S-O(0,1)` that contains the original data `0`, and another in `S-M(1,1)` that contains the new data `2`. At this point, `i4` is also retired, but is not sent to the cache system because it has already received its data from `i2`. Thus, from the perspective of the cache system and coherence protocol, the transaction with VID 3 does not contain `0xa` in its read set (i.e., `S-M(1,1)` never moved to `S-M(1,3)` as before). This could mean misspeculation is missed if for example a store occurred to `0xa` with VID 2 on another thread, and therefore the semantics of the original program are not respected.

Note that it is not a problem to forward from stores to loads if they have the same VID; this would have no impact on the HMTX coherent state of the line. Thus this is only a problem when switching between VIDs, i.e. at the boundaries of different transactions.

There are a few different ways to solve this problem. One would be to allow this forwarding to occur, but still send the load to the cache to ensure that the line updates its highVID correctly. While this mostly defeats the purpose of store-to-load forwarding entirely, it does not have a large impact on performance as changing between VIDs is relatively rare, because this only happens on the border of changing between transactions.

Alternatively, we could tag on all of the VIDs of loads that a store forwards its loads onto. However, this comes with more complexities; if there is a branch between the store and the load, then the load could be squashed if the branch is misspeculated, meaning the store would need to track what VIDs to keep track of.

Instead, HMTX takes a simpler approach: disallow such forwarding entirely. Note that this does not have much of a performance impact, as HMTX is targeted toward long running transactions, so switching between VIDs is the very uncommon case. However, this is a place where future optimizations of the HMTX design could be targeted used if HMTX is used to parallelize programs where this overhead becomes impactful.

4.8 Miss Status Holding Registers

4.8.1 Background on Miss Status Holding Registers

Miss status holding registers (MSHRs) [38] are a well known and widely used architectural technique used by non-blocking caches to keep track of outstanding misses and ensure future misses are processed correctly and efficiently.

On a normal cache miss, if there is no current prior outstanding miss for this cache line (a "primary" miss), then an MSHR entry is created and a request is sent out on the bus for this line. All future misses to this cache line prior to this primary miss being filled are considered "secondary"; these misses will see there is an outstanding primary miss for this cache line due to the existence of the primary MSHR, and so will create a secondary MSHR which waits on the already-outstanding request to be filled before attempting to satisfy its own request.

In snoopy coherent protocols like MOESI, each cache must snoop on its peers' requests to determine if it needs to respond to the request. For example, if an L1 has a line in the Owned (Dirty and not Writable) state, and it snoops a read request from a peer cache, it should respond to that read request instead of letting that request travel through to the L2

and beyond.

Therefore because MSHRs are markers of outstanding misses, they also signify what actions if any a cache should perform once an outstanding request is filled if it snooped any requests for the same line on the bus from other peer caches while the request was outstanding.

This responsibility of responding to peers' requests based on the cache's own outstanding misses (tracked as MSHRs) is known based on the design of the cache system. For example, a cache may only respond to a snooped request from a peer cache if it knows its own outstanding request which missed will be filled by the line in a Dirty state (Modified or Owned). This can only happen when the original miss was a write request.

As a more concrete example, assume some read request r_0 is outstanding for some line from a cache $L1_A$, and then $L1_A$ snoops a read request r_1 for the same line from a peer cache $L1_B$. At this point $L1_A$ knows it is not responsible for responding to r_1 from $L1_B$ once it has satisfied r_0. This is because r_0 was a read request and so the line to fill this request must be returned in a non-dirty state, and so once r_0 is filled $L1_A$ is not responsible for responding to r_1.

4.8.2 Miss Status Holding Registers in HMTX

In HMTX, it is no longer the case that a cache with an MSHR for some outstanding request knows whether it is responsible for responding to snooped requests. For example, again assume some line does not exist in some cache $L1_A$, and a read request r_0 for it has missed and a primary MSHR has been created, and then r_1 is received from $L1_B$. The line returned to $L1_A$ that satisfies r_0 may or may not be able to satisfy the response for r_1, and it is not always possible to know a priori whether it will or will not.

This is because HMTX combines many conceptual versions of a line into a single line, as discussed in Chapter 4.2, and additionally because there can be only a single instance of that line anywhere in the cache system which responds to snoops. For example, if r_0

is a read request with VID $= 2$, there are many possible instances of a line which could satisfy that request which could exist somewhere in the cache system. All r_0 needs is for its VID 2 to hit that line; that is, the line could be in `S-M/S-E(m,h)` with $2 \geq m$, or in `S-O/S-S(m,h)` with $2 \geq m$ and $2 < h$.

So if r_1 with VID $= 1$ is sent out by $L1_B$ while r_0 from $L1_A$ is waiting for a response, then $L1_A$ *must* wait for r_0 to be filled before knowing whether it can respond to r_1. For example, both `S-M(1,2)` and `S-M(2,2)` are valid possible responses for r_0. But r_1 can only be satisfied by `S-M(1,2)` and would need a response by $L1_A$, while `S-M(2,2)` cannot satisfy r_0 and would not need a response by $L1_A$.

Thus, in HMTX, $L1_A$ must conservatively keep track of r_1 in its MSHR as a potential request to respond to even though it may not actually respond to r_1. Due to the design of the protocol, there is only ever a single line in the cache system that may respond. Thus, if the line comes back for r_0 in a state that cannot satisfy r_1, then it can safely ignore r_1, knowing that some other cache or main memory will respond to r_1.

4.9 Aborting Transactions and Memory Overflow

As mentioned in Chapter 4.5, aborts should be made very rare when speculatively parallelizing a program in order to achieve a performance speedup. When it comes to parallelizing outer loops with long-running iterations of the loop made into transactions, this is even more critical when compared to the small transactions more traditionally associated with TM systems.

One kind of abort is due to invalid speculative assumptions. Some examples here are control or alias speculation that failed. This is the main motivation of TM systems; for example, to speedup multiple threads who all contain a critical section with accesses to the same memory address(es). For example, if two active transactions write to the same address, one of them must abort and retry. While this can heavily impact performance, such

a program is generally still able to complete execution, though it must strive to avoid issues with starvation and livelock. This sort of abort is seen in both software and hardware TM systems.

Another kind of abort is due to the read and write sets of a transaction not fitting inside the TM system's data structures. This is more of a meta-problem related to the ability and efficiency of the TM system to track these read and write sets. This is a problem generally seen by hardware TM systems. For example, if a TM system is designed with data structures that can only handle a specific size of read and write sets, and then a transaction attempts to speculatively access more addresses than can fit in these data structures, then the transaction must abort.

Aborts due to this lack of capacity for tracking speculative state are an entirely different problem than aborts due to alias or control misspeculation. If a single transaction has read and write sets that are simply too large to be supported by the TM system, the entire speculatively parallelized version of the program cannot make progress. That transaction must proceed non-speculatively before speculative execution can resume, or else another execution paradigm must be used.

The HMTX system is intended to enable speculative execution of very long running loops, and thus the set of memory addresses accessed is very large, up to tens of megabytes in the evaluated benchmarks (Chapter 8.4.2). However, the design of the HMTX system indeed has a limitation of the size of the read and write set. If some speculatively accessed line overflows the LLC then the transaction must abort (with some exceptions as noted in Chapter 5.4.

As noted in Chapter 10.1, an auto-tuning parallelization system could increase or decrease the amount of parallelism used by a program in order to gracefully minimize or avoid capacity aborts.

4.10 Privatized Versions of Memory

Programmers tend to reuse data structure between iterations of loops. While this can often make programming easier and/or use less memory, it can introduce spurious misspeculations in HMTX. For example if a hot loop reuses some data structure in a parallel section of a hot loop that has been speculatively parallelized, then that data structure will be accessed by many transactions at once. However, each loop iteration could use its own private version of that data structure, preventing such misspeculation.

Privatization [39, 40, 41, 42, 43, 44] is a useful way to enable parallelization in such cases, by "privatizing" data structures in order to provide each transaction its own private version in memory to work with. As mentioned in Chapter 2.3, the SMTX and DSMTX systems spawn parallel workers through process forking, thus all memory is automatically "privatized by default" via process separation, and any accesses that need speculative verification must be explicitly communicated to the commit process. Additionally, any dependences between pipelined threads must explicitly be sent via these software queues.

In contrast, the HMTX system's parallel workers are threads in the same address space, because its versioned memory is based on different versions of cache lines and their physical addresses. This model is "shared by default," meaning that all privatization needs to be explicit.

This comes with a couple benefits. First, instead of needing to rely on an extra commit process that manages all versions of memory and replays speculative memory modifications, the HMTX coherence protocol handles this without much overhead through its commit scheme discussed in Chapters 4.5 and 5.3.

Second, pipelined workers do not need to explicitly pass dependences down through software queues. Any dependences needed by a later pipelined stage from an earlier pipelined stage are naturally communicated and found during normal speculative stores and loads they share the same VID. This means a compiler or programmer parallelizing the program does not need to worry about passing these dependences explicitly.

In order for each thread to know which privatized version of memory to access, each thread must have some notion of its privatized version ID with respect to the others. This is accomplished by a single thread ID register added to the thread context. This allows for a thread to quickly determine which privatized area of memory is its own.

4.11 Operating System and Program Support

Parallelized programs must be able to query the system they are running on to determine the maximum number n of MTXs that can execute on the system at any given time. This is due to the fact that $\log_2 n$ bits are used to represent VIDs in HMTX. Once all n MTXs have been used the program must stall and take action before resuming speculative execution. This is discussed further in Chapter 5.3.2.

Additionally, the parallelized program is responsible for ensuring that the active VIDs in the system correspond back to original program order in order to ensure that correctness is maintained. It must also ensure that commits occur in consecutive order, e.g. VID 2 commits only after 1 and before 3. Otherwise, the behavior of the HMTX system is undefined.

Other requirements for transforming a program to use HMTX speculation include:

- If a program needs to perform allocation during speculative sections of the program, it must use some pre-allocated thread-local chunk of memory, for example nedmalloc [45][2]. Otherwise there will be false misspeculations forced through calls to `sys_brk`.
- Inputs and outputs must be handled specially inside a transaction. Inputs must be read non-speculatively either before or during a hot loop and potentially rewound if misspeculation occurs, while outputs must be explicitly buffered to ensure no specu-

[2]Note that the chunks of memory pre-allocated for nedmalloc should be aligned to cache line boundaries; otherwise accesses to memory from separate transactions may end up interacting and cause false misspeculation.

lative outputs occur until commit.

- Data structures accessed during speculative sections of the program must be privatized, as discussed in Chapter 4.10.
- Any thread performing speculative operations should not exit until all transactions they participated in commits have occurred. Otherwise their stacks (which are marked as speculative) will be reclaimed by the operating system, causing misspeculation.

Chapter 5

Supporting and Optimizing For Complex, Long-Running Transactions

Most existing HTM systems do not provide sufficient support for long-running and complex transactions that are often required for long-running and complex programs. This section introduces important enabling optimizations that prevent false misspeculation due to branch-speculative loads getting squashed (Chapter 5.1); allow for interrupts and exceptions without forcing aborts (Chapter 5.2); enable extremely fast and simple atomic commits across the cache hierarchy (Chapter 5.3); and allow for certain speculatively read lines to overflow the last level cache without forcing a conservative abort (Chapter 5.4).

5.1 Squashed Loads and False Misspeculation

As discussed in Chapter 3.4, cache lines may become incorrectly marked as speculatively accessed due to branch misprediction inside of a transaction, leading to spurious misspeculations.

For example, assume some load from address `0xa` with VID `1` occurs, where the line containing `0xa` has not yet been speculatively accessed at all. When sent to the cache, the request will mark the line containing `0xa` as having been speculatively read by VID

1. However, if that load is squashed because branch prediction was incorrect, then the line has been incorrectly marked as speculatively read, which may cause false misspeculation. Note that this is a performance issue and not a correctness issue.

To overcome this problem, the *speculative load acknowledgment* (SLA) is introduced. When a branch-speculative load is executed it does not immediately mark the line it accesses with its VID. Once the load is actually committed, then it is safe to mark the line with its VID. At this point an SLA is sent to the cache system, which includes the value which was loaded, the address of the load, and the VID of the load. A structure similar to the store queue buffers these SLAs until they should be sent. The cache system receives this request, verifies that the original value loaded in the SLA is the same as the current one at that address, and then transitions the line to the correct speculative state. Otherwise an abort is triggered. Depending on the parallelized program, this optimization may avoid many false misspeculations (Chapter 8.4.1).

Note that this does not cause a race condition between transactions that can cause incorrectness. This essentially defers the update of the VID of the load to the line in the cache. However SLAs only impact correctness for interactions between accesses of different VIDs, the ordering of which are naturally tracked through the HMTX VID scheme. For example, if a line is speculatively read by some transaction with VID x and some other transaction with VID y stores to this same location, the temporal order of these accesses does not matter, including whether and when an SLA is sent. If $x < y$ then the store should always succeed without any issue. If $x > y$ then if x occurs temporally first then the store for y will cause an abort because of a read-after-write violation. If y occurs temporally first then the value written should flow correctly into the load for x. This all holds regardless of whether the load for x required an SLA.

Importantly, SLAs may not always be necessary to send. For example, an SLA does not need to be sent for an access to a line that already has logged that the VID accessed it, either from an earlier confirmed speculative load with the same VID, or from a speculative

store with the same VID. Thus, to ensure sending SLAs does not greatly negatively impact performance, when a speculative load executes, it is returned with a bit representing if an SLA is required. This way when the load is committed due to correct branch prediction it knows whether or not to send an SLA. Thanks to memory access locality, the number of SLAs that need to be sent is low (Chapter 8.4.1).

SLAs have some similar characteristics to compare-and-swap (CAS) instructions. A CAS instruction takes three operands: an address, an old value, and a new value. If the value at the address matches the old value, then the new value should be stored, and otherwise the new value is not stored. This is similar to SLAs in that the originally read value is stored in the SLA queue along with an address, and once the SLA is sent it checks that the value at the address is the same as the value in the SLA (compare). If so then the line is marked with the VID of the SLA. Otherwise, the transaction is aborted.

Similar to CAS instructions, SLAs are susceptible to the ABA [46] problem. This is where some original value A from a memory location is saved somewhere (e.g. in a register in the core), for example for a CAS or SLA to use. The memory location is then modified to some new value B, and then changed back to A before the CAS or SLA has a chance to check its stored version A against the new version. Then the CAS or SLA checks the value and observes the value is still A, even though it was in fact changed to B and back to A.

Note that this problem is not something that can occur in HMTX as long as any given VID is only in use by a single thread at a time, which is the case for all benchmarks evaluated in this dissertation. If the intermediate modifications occur from some thread using a different VID then they access another version of memory, and so the value cannot be "silently" modified.

Still, if some use case desires to use a single VID by multiple threads concurrently, the ABA is a well studied problem and, similar to CAS, could be solved by for example by adding some extra meta information (perhaps in unused bits in the address) like the number of times the address has been modified to check that the value has in fact not been changed.

5.2 Surviving Interrupts and Exceptions

In order for transactions to survive interrupts (e.g. for context switches) and exceptions (e.g. for virtual memory management), the operating system must be able to non-speculatively perform memory operations once interrupted. This is especially important for long running transactions and those which access large or irregular pointer-chasing data structures. Even programs without such memory access patterns may require non-speculative exception handling, as operating systems often lazily load pages on-demand and thus would need to non-speculatively ensure all required memory inside a transaction is already loaded prior to speculative execution.

Note that all such non-speculative actions do not result in actions that impact correctness of the system as a whole. Context switches are a common and natural part of execution of a program; the fact that a speculative program is switched out should not be a reason to abort a program, and causes no issues of correctness. Similarly, moving entries around in the page table of a process does not impact correctness of other processes or the system as a whole. Therefore it is safe to do these operations non-speculatively while in the middle of speculative execution.

To support this, the parallelized programs are statically linked, and then the program informs the HMTX system of the range of the program's text segment, so that it will only add the VID onto loads and stores that fall into this PC range. This results in functioning non-speculative interrupts. Dynamically linked programs could also be supported if the system is made aware of the addresses of the libraries.

This is implemented via two speculative PC range registers added to the core. When an instruction is fetched its PC is compared against these registers; if the PC falls into the range then this instruction should be considered MTX-speculative, and the VID set in the VID register is appended to the instruction if it's a memory instruction.

Additionally, note that unlike most hardware TM systems, speculative threads can migrate between cores; their data can be found in other caches naturally through the VID

of the transaction. Thus the OS is free to interrupt and move a thread participating in a transaction from one core to another without issue.

5.3 Commit and Abort Handling

As noted in Chapter 3.4, a naïve scheme such as that presented in Chapter 4.5 may need to keep track of all speculatively accessed lines in order to explicitly transition them on commit, or else use a lot of time or hardware complexity to commit them quickly. This could require some structure in hardware or software to scale along with the number of accessed lines (which is quite large in the evaluated benchmarks (Chapter 8.4.2)), increasing complexity and degrading performance of the system in execution time and energy.

For example, something similar to an ownership required buffer (ORB) [15] could be used, which simply holds the addresses of all lines for every speculative access. Processing this buffer would take a significant amount of time as it would need to be searched and cleared for every address. This would need to happen for all caches that have speculative state, which in the evaluation includes a large last level cache.

To improve upon this scheme, the HMTX system is able to take advantage of the design of the protocol, as well as adapts an approach used by other works [18, 34, 47] by using *lazy commit processing*.

5.3.1 Efficient per Commit Action

To efficiently handle commits, a new register is added to each cache representing the latest committed VID (LCVID). On commit for some commitVID, all caches simply set their LCVIDs to commitVID. This is the *only action required* on commit. This is done atomically across the system once broadcasted on the shared bus between the L1 and L2.

For every speculative access that arrives, the cache continues using the same hit and miss logic as before, as described in Chapter 4.1. Non-speculative accesses (those with

VID $= 0$) however instead use VID $=$ LCVID just for determining if there is a hit. This intuitively makes sense, as non-speculative accesses should access the last committed version of a line.

When a line is selected as a victim for eviction to the next level cache, it can be lazily transitioned before sending to the next level if necessary (i.e. it may have been invalidated). It may additionally be lazily processed after a VID reset, discussed in the next section.

5.3.2 Lazy Commit Processing via VID Overflow and Reset

VIDs are limited to a finite m bits. When running many iterations of a loop speculatively with HMTX, it is likely that we will eventually commit $2^m - 1$ transactions, meaning we run out of VIDs. All transactions from a single group of $2^m - 1$ transactions (i.e. transactions with VIDs from $1 \rightarrow 2^m - 1$) are called a "flight" of transactions.

The parallelized program needs to be able to somehow query the system to determine this $2^m - 1$ maximum number of VIDs that constitute a flight, as mentioned in Chapter 4.11.

Once the last transaction of a flight commits, first the software must delay all new transactions until the one with VID $= 2^m$ has committed. Next, a VID Reset signal should be sent to the memory system, which triggers two actions once broadcasted on the shared L1-L2 bus:

1. All cache lines set their VIDs to `(0,0)`.
2. All caches set `LCVID` $= 0$.

Once this is complete, a new flight can begin, with the first transaction of the flight starting again with VID $= 1$. Note that only one flight can be active at a time, and all transactions from a flight must be successfully committed before that flight is retired and a new flight can begin.

This works thanks to the design of the protocol (Chapter 4.1); the combination of these three actions results in following the exact commit state diagram from Figure 4.4:

- The "latest" versions of the line in `S-M/S-E` will now have VIDs `(0,0)`. This

effectively commits them even though they are not yet moved to M/E, because they have their modVID set to non-speculative state (i.e. $== 0$), and their hit condition simply checks for the request's VID ≥ 0, which will always be true regardless of whether the request is speculative or not. The next time this line is accessed, the line lazily sets SB $= 0$, which officially moves S-M to M and S-E to E.

- The "non-latest" speculative versions of the line in S-O/S-S(0,0) can never hit for an access because their hit condition checks for the access VID < 0 which can never be true. Thus they are effectively invalidated. These lines will simply be invalidated when selected as a victim for eviction.

This elegantly and quickly allows for commits to proceed and for VID overflows to occur without requiring very much complexity in hardware or software.

Note that when using DSWP, VID Resets can be costly because they stall the DSWP pipeline until the transaction with maximum VID $= 2^m$ commits. Thus the decision of how many bits should be used for VIDs leaves a tradeoff of execution time vs. implementation complexity and energy consumption. 6 was settled on as a fair compromise.

5.3.3 Abort Processing

As discussed in Chapter 4.5, on an abort for any VID, all uncommitted transactional memory in the cache system is flushed.

When an abort occurs, it is broadcast on the shared L1-L2 bus. Then, for each line, if SB is set , then it is transitioned according to the state diagram in Figure 4.5 with one change: instead of checking if modVID $== 0$ and modVID > 0, it checks if the line would be hit by LCVID or not, respectively.

For example, assume the cache had lines in S-O(0,2), S-O(2,5), and S-M(5,7), and LCVID $= 3$. Following the above logic, S-O(0,2) and S-M(5,7) would not be hit by VID 3 and so would be invalidated based on the abort state diagram. Meanwhile, S-O(2,5) would be hit by VID 3, and so would move to M based on the abort state

diagram.

Note that all lines from previously committed flights have already been committed as discussed in Chapter 5.3.2, and so the abort logic does not impact these lines here. Additionally, because the condition for determining if something has been committed is based on the hit logic with LCVID, all lines already committed based on the current value of LCVID are also left as committed, while other lines are invalidated.

Additionally, note that the hit logic needed for determining how to proceed on an abort already exists in the cache for normal hit and miss processing. However, cache sets cannot all be checked at once. Hence the abort transition will take an amount of time that scales with the number of cache lines in the cache. This is an intentional design choice because aborts should be extremely rare, and so we limit the complexity of the system and push the common case commit to the fast path while allowing the rare abort case to take longer.

5.3.4 Summary of Commit and Abort Design

This design includes limited complexity for cache operations on commit or abort. On the common-case path of commits and unavoidable VID overflow, execution can proceed quickly without waiting for an expensive or costly cache operation where, all at once, every line must be explicitly transition based on the commit or abort state machine.

This scheme is made possible by the design of the coherence states, which allow for a line to transition to its next state without needing to query for the state of any other lines in the system, and due to the elegance of resetting VIDs to `(0,0)` effectively performing a commit. This commits all lines from a completed flight, while the LCVID enables understanding of what has been committed in a partially completed flight.

Summarizing the protocol for commits, completed flights, and aborts:

- On commit of a single transaction inside a flight, simply set all caches' LCVID to the VID of the commit. This keeps commits on the fast path and makes their implementation and cost very low.

- On a completed flight, software pauses speculative execution and:
  1. All cache lines set their VIDs to `(0,0)`. This effectively commits and transitions all lines from all transactions in the flight.
  2. All caches set `LCVID` $= 0$. This allows for a new flight to begin back at VID $= 1$.
- On abort, every cache line checks if LCVID hits the line to determine how to transition the line for abort. Checking for a hit allows for the correct transition to be made in an equivalent fashion to the abort state diagram in Figure 4.5.

5.4 Speculative Memory Overflowing the Caches

Each cache line's modVID and highVID enable different versions of memory as well as tracking and verifying the ordering of accesses to these versions. Naïvely, this means that all speculatively accessed lines would need to stay inside the caches for the system to function correctly.

Note however that the scheme saves many non-speculative versions (in `S-O` with modVID $= 0$). Because these lines are non-speculative, they are safe to write back to memory. However, it must be guaranteed that they can be retrieved back in a speculative state that ensures correct execution. The protocol ensures this because if there is a line in `S-O`, there must also be an `S-M` line also somewhere in the cache hierarchy (as seen in Table 4.3). If an `S-M` line snoops a request with VID $=$ `y` for a line that has the same address but does not hit due to VID comparison, it asserts that the line was already speculatively modified. If the request then misses all caches, it knows that it should have hit an `S-O` version with modVID $= 0$ that must have been written back to memory. Thus when the request is satisfied by memory it is returned in `S-O(0,y+1)`, and speculative execution can resume. This preserves correctness while allowing for larger read and write sets.

Chapter 6

Complete Design Overview

This Chapter provides a review of all of the architectural changes required for HMTX. Figure 6.1 shows a graphical depiction of these changes.

Each core is augmented with:

- A VID register (Chapter 3.2), which represents the current active VID to use for all memory operations.
- Two speculative PC range registers (one low and one high) (Chapter 5.2), which represent the range of PCs from which memory instructions should have the VID register added to the memory operation, allowing for interrupts and exceptions to be handled.
- Speculative Load Acknowledgment (SLA) Queue (Chapter 5.1), used for sending confirmation when a branch-speculative load that is also MTX speculative is committed.

New HMTX instructions are added to the ISA (as discussed in Table 3.1):

- `MTX_INIT(handlerFunction)`: Registers a fallback abort handler function. Should be called by all speculative threads prior to speculative execution beginning.
- `MTX_BEGIN(vid)`: Marks the beginning location of using some VID for all memory operations that follow. Sets the VID register in the core to `vid`.

CacheVIDs = (ModifierVID, HighestAccessorVID)
ModifierVID = Transaction which created this version of the line
HighestAccessorVID = "Highest" transaction to access this line

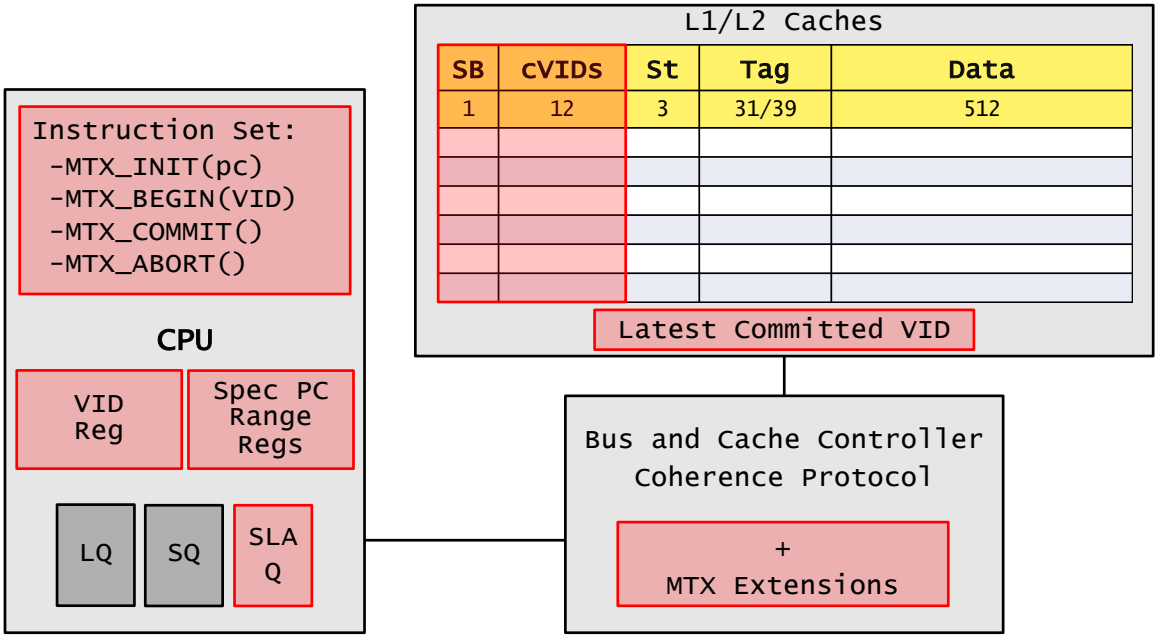


Figure 6.1: Final Overview. Those items highlighted in red are added for HMTX. St represents each line's coherent status before addition of the Speculative Bit `SB`.

- `MTX_COMMIT()`: Commits the currently active transaction, based on the most recently entered transaction via `MTX_BEGIN`.
- `MTX_ABORT()`: Aborts all active transactions. All speculative threads will fall back to the handler function specified by `MTX_INIT(handlerFunction)`.

Caches include support for multiple versions of lines based on VIDs. This must be done at least for the closest level cache to the core, and can be expanded to the next level after in order to provide for larger transaction sizes. In the presented implementation, this includes both the L1 and L2 caches.

Each cache includes a Latest committed VID (`LCVID`) (Chapter 5.3.1), used for representing the latest VID to have been committed in the current flight of transactions.

For every line in these caches, the following is added (visualized in Figure 6.2):

- Cache line VIDs are added to every line, which is a tuple of the `(modVID,`

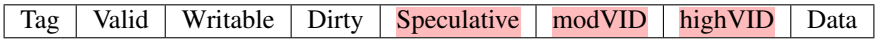


Figure 6.2: Structure of a cache line. Those components highlighted in red are part of the HMTX design. In total, 13 bits are added per line: 1 bit for the Speculative bit, and 6 bits each for modVID and highVID.

`highVID`), each 6 bits in the presented implementation (Chapter 4.2):

  - Modifier VID (`modVID`) represents the the transaction which created this version of the line.
  - Highest Accessor VID (`highVID`) represents the "latest" transaction to have accessed this line.

- Speculative bit representing if this line is in a speculative coherent state (Chapter 4.1).

Using this state on a per cache line basis, logic is added to track the coherent state of lines as they receive accesses and snoop requests on the bus. Comparators must be added alongside tag comparisons (e.g. the cascading comparators mentioned in Chapter 4.6). The state machine for the traditional MOESI coherence model is extended to include the transitions described in Chapter 4.1.

Chapter 7

Preserving Original Program Semantics

This Chapter provides a discussion of the correctness of the HMTX system. Chapter 7.1 provides an argument that, given some parallel program that uses HMTX as intended, the program behaves as described in this dissertation and respects data hazards given some original sequential ordering of the program mapped to HMTX with VIDs. Chapter 7.2 lays out an exhaustive analysis of all possible coherence states the system can move through given two requests, and how those requests respect data hazards. Chapter 7.3 outlines how HMTX would be integrated with an ISA like Alpha which uses a weakly-consistent memory model.

7.1 Argument For Respecting Original Program Data Hazards

Parallel transformation of a sequential program to use versioned memory using HMTX is essentially assigning each dynamic instruction of that program a VID, and then executing each on one or more threads with correct synchronization between transactions and stages.

The presented analysis assumes that this transformation and assignment is done correctly, i.e. that VIDs are assigned in order corresponding to the original sequential ex-

ecution order of the program, and thus they can be used to respect dependences during speculative execution, ensuring the original program's semantics are preserved.

This section focuses on the interactions between inter-transaction loads and stores. Note that while a thread is using a particular VID, no protections against data races are provided if another thread is concurrently using that VID. In the programs evaluated in this dissertation, a VID is only in use by a single thread at a time. Intra-transaction operations can assume the same semantics as a normal non-versioned memory system. Further analysis is provided in Chapter7.2.

Inter-transaction operations require more reasoning than intra-transaction operations. Given two transactions with VIDs x and y where $x < y$, all memory operations with VID x should occur logically before those with VID y, as x represents some dynamic sequence of instructions that came before instructions with VID y in the original sequential program. If this logical order is not respected during execution then then an abort should be triggered.

Assume for the following cases that the system receives two memory operations, at least one of which is a store, with VIDs x and y with no intervening accesses. Additionally assume that their VIDs are the highest two VIDs to access this line; if this were not the case, then some version of the line would have highVID $>$ the VID of at least one store, which would trigger misspeculation.

True Dependences (read-after-write): Assume a store with VID x, s_x, to the same address as a load with VID y, l_y. The value stored by s_x should be loaded by l_y, or else an abort should be triggered.

- **Dependence Respected:** If s_x occurs temporally first, then the version of the line with this modification will exist in `S-M(x,x)`. When l_y occurs, it will hit this version of the line because $y > x$, and the line will move to `S-M(x,y)`. Thus the correct value will be loaded thanks to uncommitted value forwarding.
- **Dependence Not Respected:** If l_y occurs temporally first, then a version of the line must exist with highVID $== y$ in either `S-M`, `S-E`, or `S-O`. When s_x occurs mis-

speculation will be detected because of a store to a line with VID $<$ the highVID of the line, y, and an abort will be triggered because the dependence was not respected.

Anti-Dependences (write-after-read): Assume a load with VID x, l_x, to the same address as a store with VID y, s_y. The value stored by s_y should not interfere or replace the value loaded by l_x.

- **Dependence Respected:** If l_x occurs temporally first, then a version of the line will exist with highVID $==$ x in either `S-M`, `S-E`, or `S-O`. Note that if `S-O`, there must exist some other `S-M` with modVID $==$ y from a store that occurred earlier; otherwise if `S-M` or `S-E`, modVID must be $\leq$ x. Then when s_y occurs, it will hit an `S-M` or `S-E` of the line because one of these lines must exist with modVID $=$ x or y. If the line hit was not `S-M(y,y)`, then a new copy of the line will be created in `S-M(y,y)`, and the original version of the line will be left in `S-O` with highVID y.
- **Dependence Respected:** If s_y occurs temporally first, then a version of the line will be created or already exist in `S-M(y,y)`, and then some other line in `S-O` with highVID $==$ y will exist with modVID $<$ y. When l_x occurs it will hit this `S-O` version of the line because $x < y$. The correct value will be loaded and false misspeculation will be avoided.

Output Dependences (write-after-write): Assume two stores with VIDs x, s_x, and y, s_y.

- **Dependence Respected:** If s_x occurs temporally before s_y, then the version of the line with this modification will exist in `S-M(x,x)`. When s_y occurs, it will hit this version of the line because $y >$ the modVID of the line, x. A new copy of the line will be created in `S-M(y,y)`, and the original version of the line will be left in `S-O(x,y)`. Two versions of the line will exist and false misspeculation is avoided.
- **Dependence Not Respected:** If s_y occurs temporally before s_x, then a version of the line will be created in `S-M(y,y)`, and some line in `S-O` must exist with highVID $==$ y. When s_x occurs misspeculation will be conservatively triggered because of a

store to this S−O version of the line. Note that this is necessary because of the cache line granularity of tracking speculative accesses. If s_x writes to a different part of the line than s_y, and some read occurs with VID y after s_y to the location s_x wrote to, then that read would not read the correct value.

Thus, all dependences will be respected, and the original program's sequential semantics will be preserved.

7.2 Exhaustive Analysis of All Possible States

An exhaustive analysis is presented in this section. In general, there are many dimensions to consider when determining what may happen between two accesses[1]:

- Whether the accesses are reads or writes
- Whether the accesses have the same or different VIDs
- The order of the accesses

Six figures are presented (Figure 7.1 through Figure 7.6) enumerating all possible combinations of interactions for two accesses given these possibilities. The following represents all such combinations, where r and w represent reads and writes respectively, and the subscript represents the VID of the access:

- Two accesses using the same VID, i.e. VID a then a:
  - w_a, r_a (Figure 7.1)
  - w_a, w_a (Figure 7.1)
  - r_a, r_a (Figure 7.2)
  - r_a, w_a (Figure 7.2)
- Correct ordering of two accesses based on VID, i.e. VID a then b:

[1]Note that in addition, these accesses could come from the same or different threads. To limit the search space, threads are abstracted away from this analysis. The design of HMTX limits modifications to S−M and S−E lines, of which there can only exist one in the system for a specific cache line. If a write request needs access to one of these lines it will request it on the bus an get it. Thus, when discussing any two accesses, it is assumed the lines could exist anywhere in the cache system, and the accesses could come into any L1 in the cache system.

  - w_a, r_b (Figure 7.3)
  - w_a, w_b (Figure 7.3)
  - r_a, r_b (Figure 7.4)
  - r_a, w_b (Figure 7.4)
- Incorrect ordering of two accesses based on VID, i.e. VID b then a:
  - w_b, r_a (Figure 7.5)
  - w_b, w_a (Figure 7.5)
  - r_b, r_a (Figure 7.6)
  - r_b, w_a (Figure 7.6)

Given this exhaustive analysis, it is possible to see that all data hazards are identified and either all dependences are preserved, or else an abort is triggered.

For example, Figure 7.3 depicts a write request with VID a followed by two possible secondary requests: a write with VID b, or a read with VID b. The yellow section on the left enumerates all possible initial cache states. The green section in the middle enumerates all possible resulting cache states given a write with VID a. The blue section in the upper right enumerates final states given a write with VID b, while the red section in the lower right enumerates final states if instead the second request is a read with VID a.

Each block consisting of one or more lines represents one possible cache state; the lines in that block could exist anywhere in the cache system, and if a request misses in its local cache that line will be the one that hits for this request. It will be transferred to the L1 initially which received the request.

For example, `S-M(<a,≤a)` is one line in a possible cache state represented by its containing block. The block includes one or more other lines in `S-O`, including one with modVID `0`. Given this cache state, if a write was received somewhere in the cache with VID a for value `0x8`, then the resulting cache state would include `S-M(a,a)` with the new data `0x8`, as well as the previous cache line data in `S-M(<a,a)` with the old data `0x2`. From there a read (red arrows/section) or write (blue arrows/section) with VID b is

shown to occur with each of their possible resulting cache states. If it's a write then again we have a new `S-M(b,b)` line with the new data `0x9` as well as the old line in `S-O(a,b)` with the old data `0x8`. In contrast if it's a read then the `S-M` line simply updates highVID, resulting in `S-M(a,b)` line with the same data `0x8`.

Note that no S-S lines are used in this analysis. This is because these lines are always unmodifiable copies of other lines in the cache, and can only represent some "frozen" version of a line (i.e. it can never represent the "latest" version and thus no writes can be performed to them, and no reads can occur that wouldn't occur to the version of the line that they came from). For example, if a line in `S-M(a,b)` spawns a `S-S(a,b)` version, the hit logic for `S-S` prevents requests with VID b from hitting this line, and thus it can only serve as a base for reading already frozen lines which cannot be speculatively modified without an abort being triggered. Thus they respond to reads between $\geq a$ and $< b$ just as the `S-M(a,b)` line would, and otherwise requests $\geq b$ will miss and eventually hit the `S-M(a,b)` line.

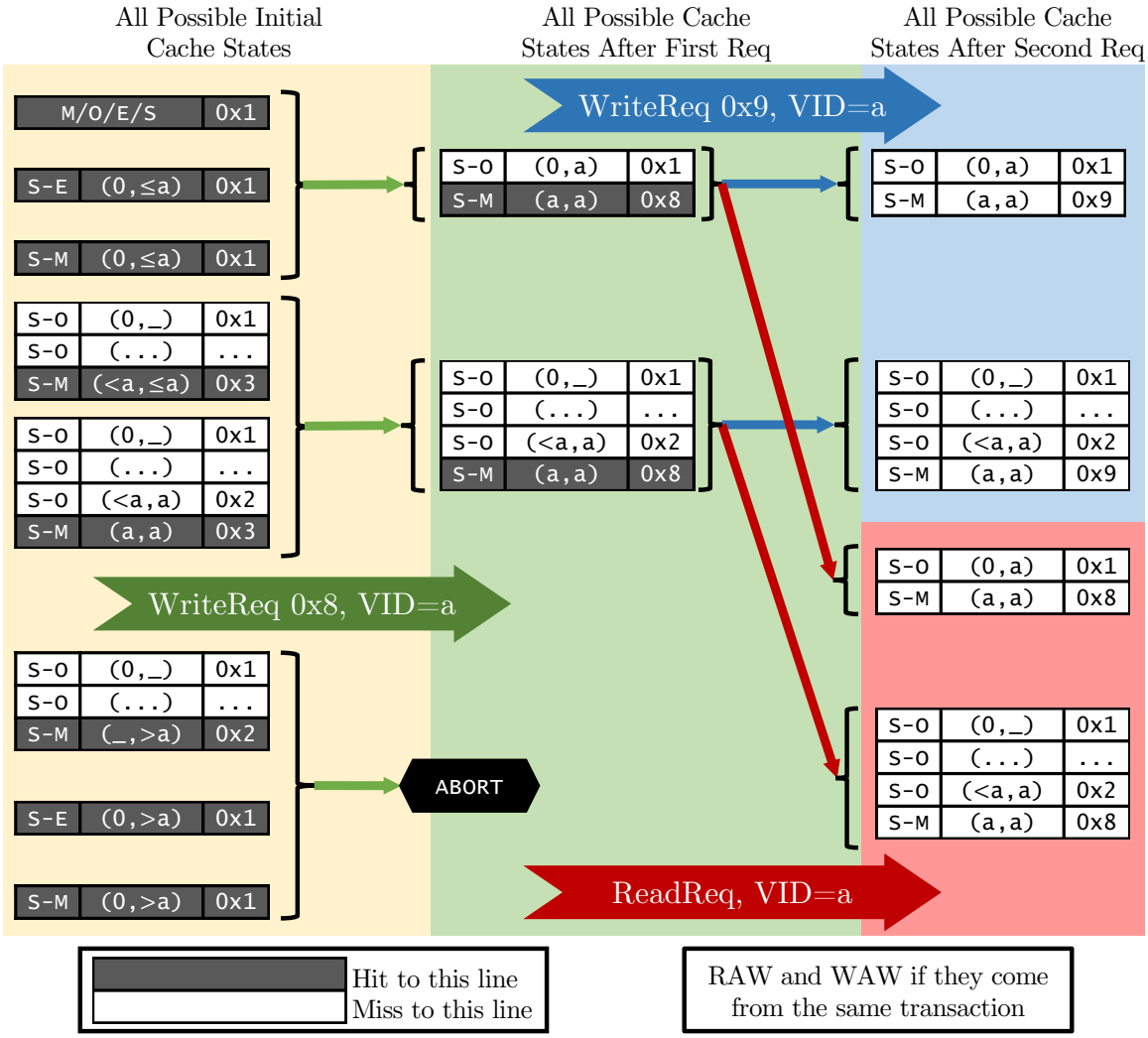


Figure 7.1: Depiction of all possible cache states as the system receives two requests with the same VID. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a write request with VID a. Given these new states, the blue section in the upper right represents all possible cache states after another write request with VID a. The red section in the lower right represents all possible cache states after a read request with VID a instead of a write request.

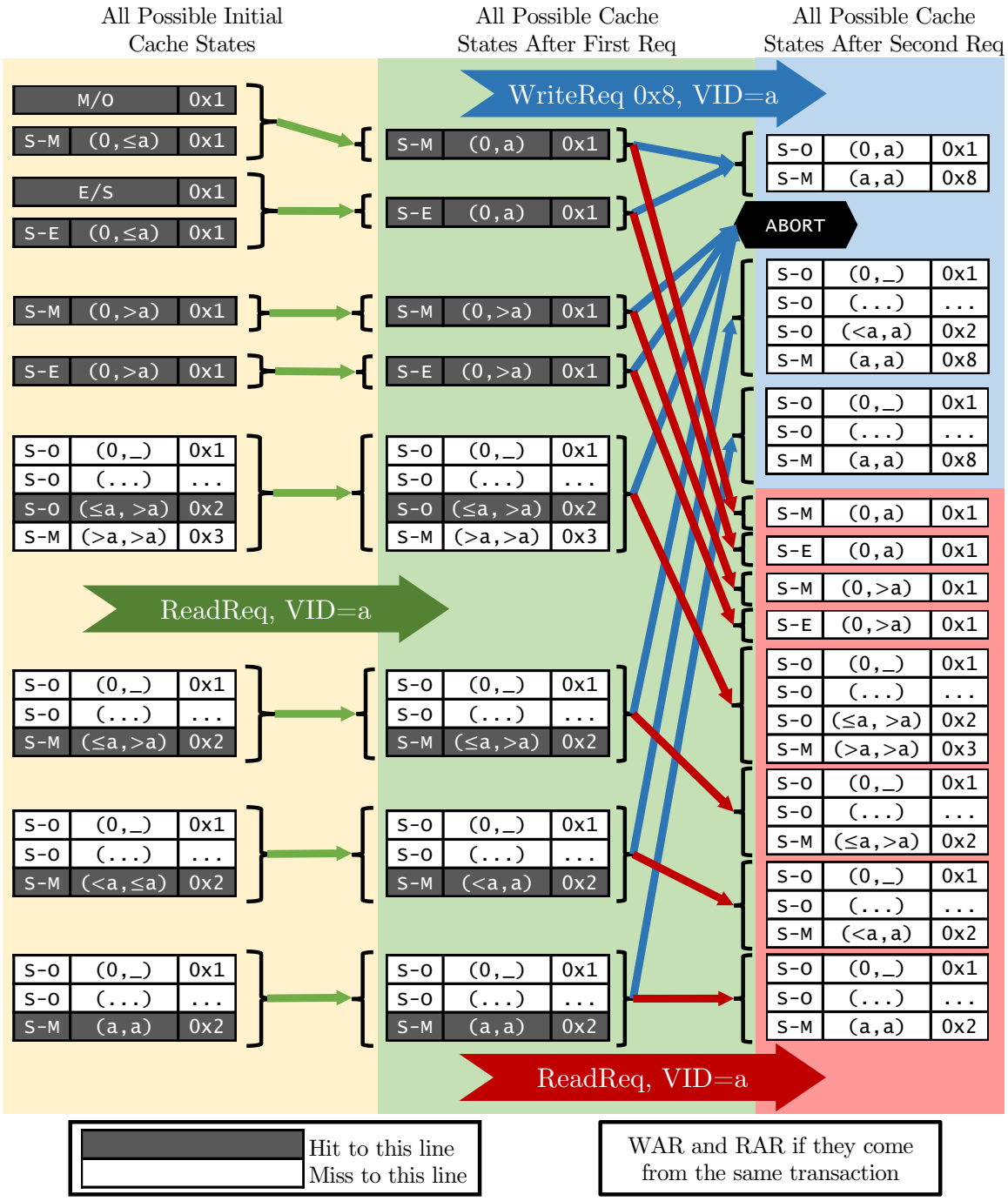


Figure 7.2: Depiction of all possible cache states as the system receives two requests with the same VID. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a read request with VID a. Given these new states, the blue section in the upper right represents all possible cache states after a write request with VID a. The red section in the lower right represents all possible cache states after a read request with VID a instead of a write request.

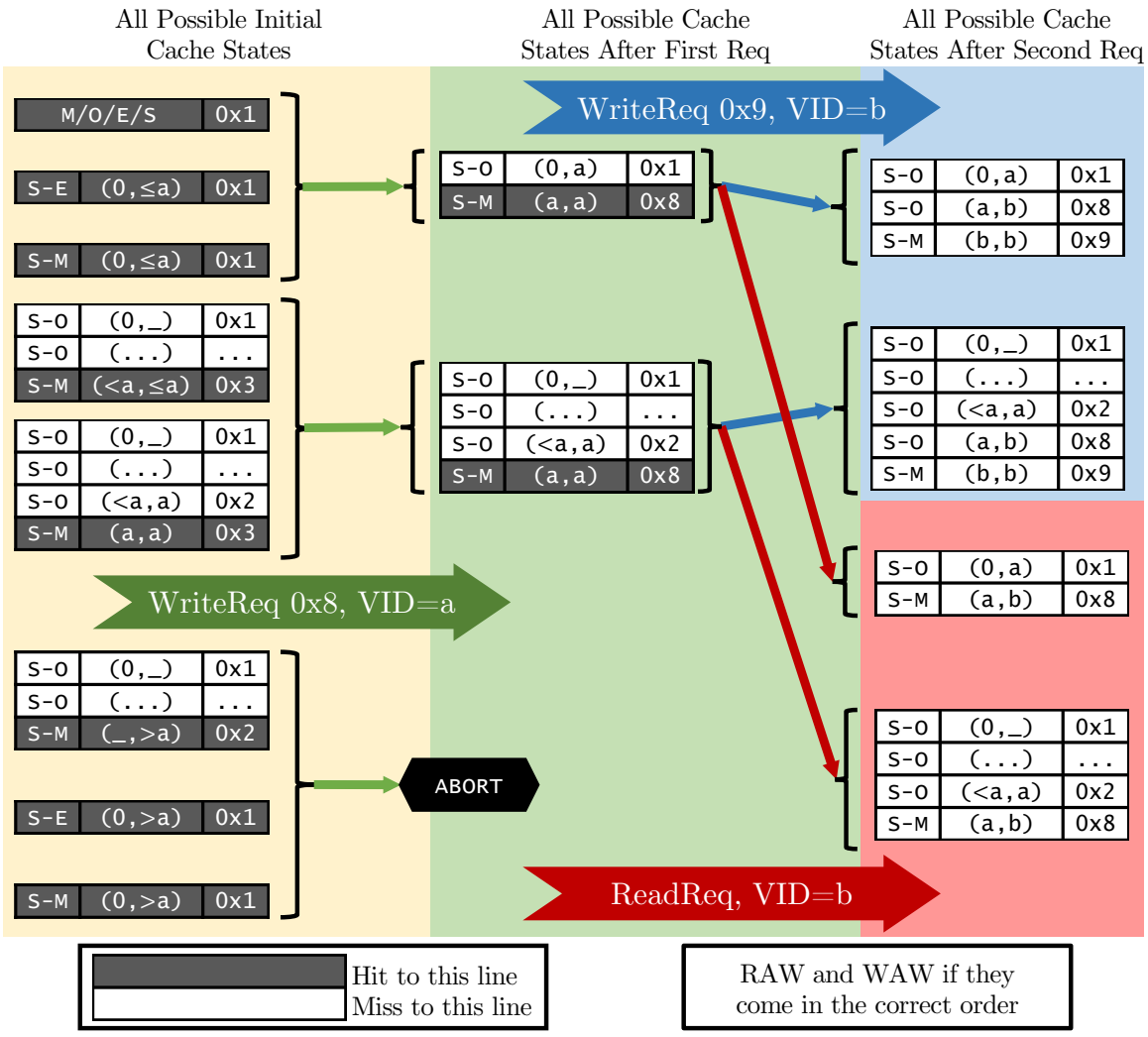


Figure 7.3: Depiction of all possible cache states as the system receives two requests with VIDs a and b in that order. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a write request with VID a. Given these new states, the blue section in the upper right represents all possible cache states after write request with VID b. The red section in the lower right represents all possible cache states after a read request with VID b instead of a write request.

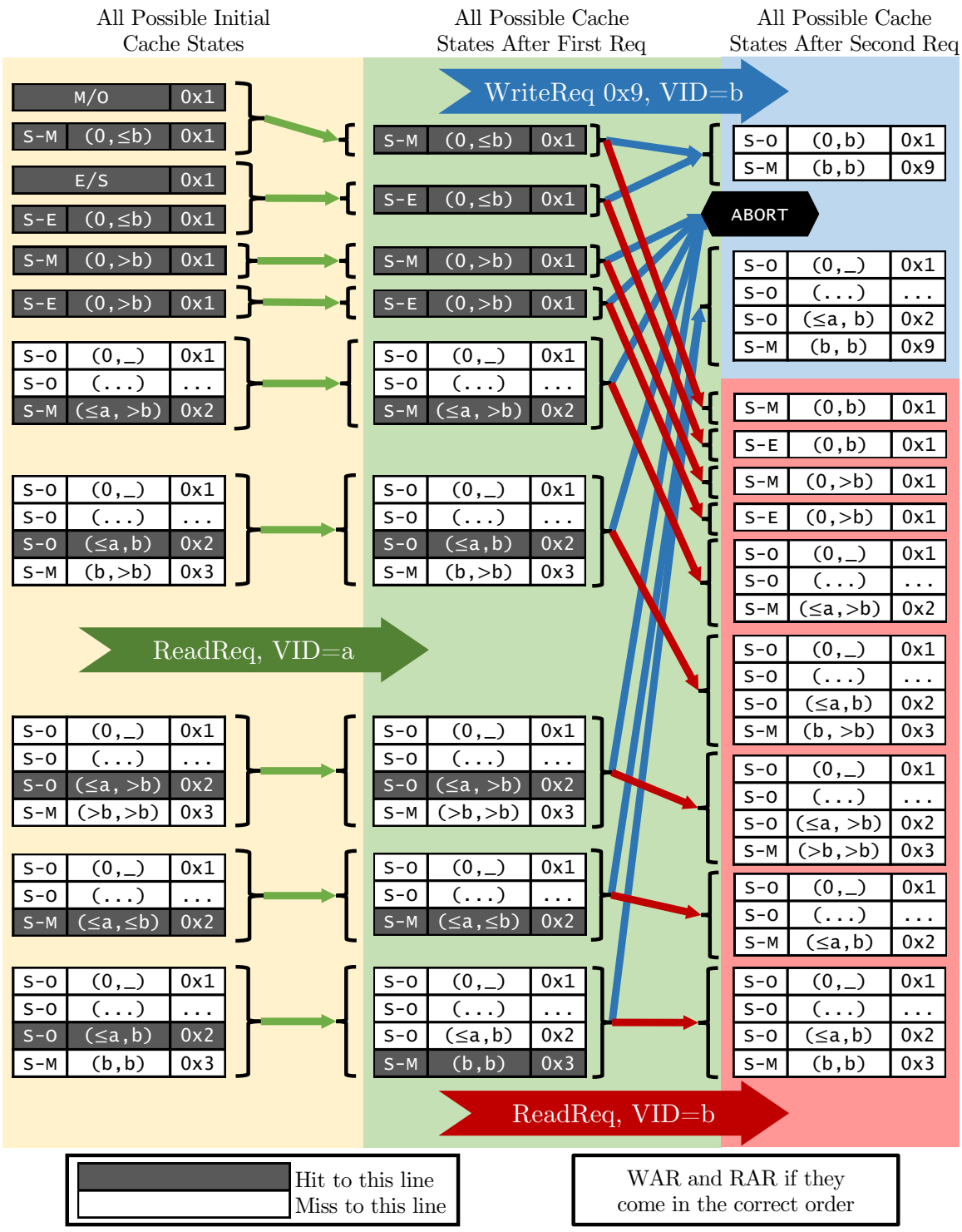


Figure 7.4: Depiction of all possible cache states as the system receives two requests with VIDs a and b in that order. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a read request with VID a. Given these new states, the blue section in the upper right represents all possible cache states after write request with VID b. The red section in the lower right represents all possible cache states after a read request with VID b instead of a write request.

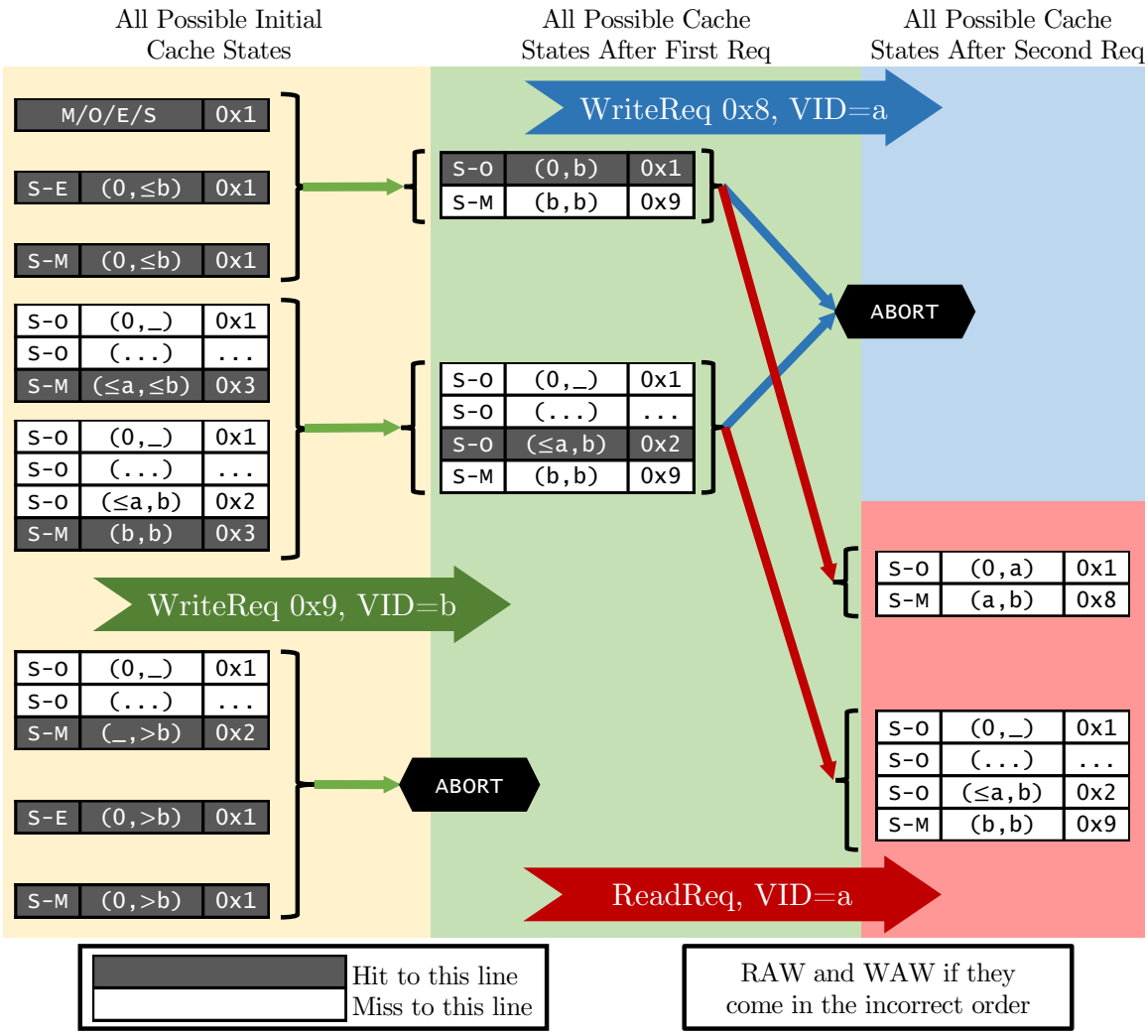


Figure 7.5: Depiction of all possible cache states as the system receives two requests with VIDs b and a in that order. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a write request with VID b. Given these new states, the blue section in the upper right represents all possible cache states after write request with VID a. The red section in the lower right represents all possible cache states after a read request with VID a instead of a write request.

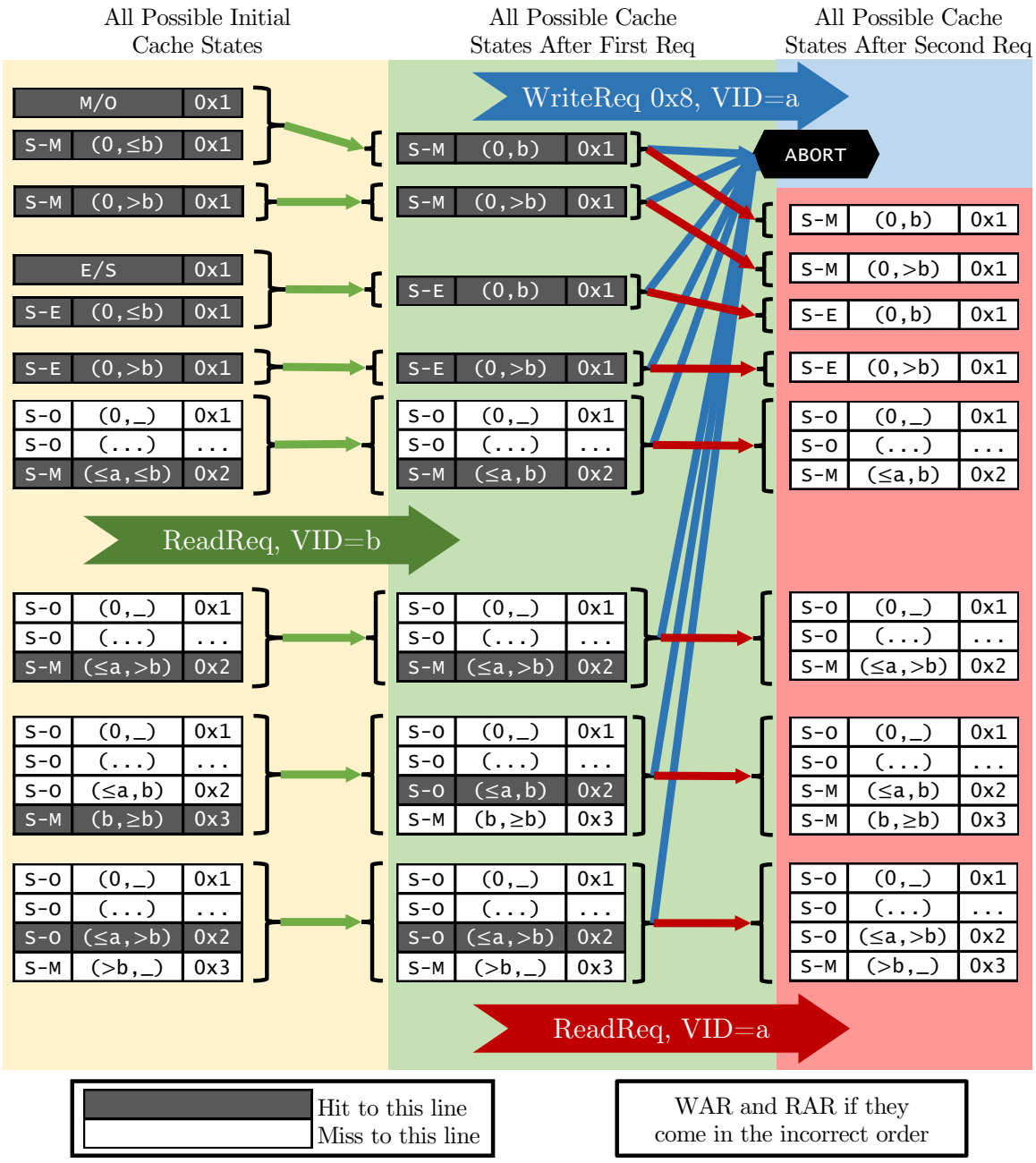


Figure 7.6: Depiction of all possible cache states as the system receives two requests with VIDs b and a in that order. The yellow section on the far left represents all possible initial cache states. The green section in the middle represents all possible cache after a read request with VID b. Given these new states, the blue section in the upper right represents all possible cache states after write request with VID a. The red section in the lower right represents all possible cache states after a read request with VID a instead of a write request.

7.3 Memory Consistency Model and HMTX Speculative Memory

HMTX was modeled using the Alpha ISA, which has a weakly-consistent memory model. Accordingly, synchronization barrier instructions are required to enforce an ordering in specific circumstances to ensure a bug-free program. This must be considered in the context of buffered transactional memory state and how and when this state becomes visible to other threads on other cores.

Similar to the discussion in Section 4 of [48] about transactions in the Power ISA which contains "Load and Reserve" (larx) and "Store Conditional" (stcx) instructions, Alpha contains load-locked and store-conditional instructions. As discussed in [48], these instructions' correctness relies on the fact that a pair of these instructions will fail even if the load reads a stale value, i.e. some other store has already occurred elsewhere in the system but the value read by the load has not yet been updated.

This concept is then related to implementing transactional memory in a weakly-consistent ISA like Power; they note:

> "[A] transactional store must gain control of the coherence mechanism for the given location to ensure that the transactional write will enter the coherence order for the location as the most recent value, and that no store by another thread will subsequently be serialized into the coherence order for that location before the transaction commits. This is typically achieved by gaining write ownership of the location in the coherence protocol and either holding that ownership until the transaction commits successfully or failing to hold that ownership due to a conflict with another transactional or non-transactional access and failing the transaction.
>
> Additionally, a major difference between transactional semantics and larx/stcx semantics is that, unlike larx/stcx pairs for which the stcx can succeed once

> the write has been serialized into the coherence order and before the write has propagated to all processors to eliminate stale copies, the transaction must propagate (or at least appear to propagate) to all processors and make any stale cached copies of the locations un-readable before a transaction can successfully commit. This is necessary to prevent non-transactional loads on other threads observing an inconsistent image of the aggregate store."

A similar argument can be made for HMTX using the Alpha ISA which has its load-locked and store-conditional instructions. HMTX requires that any cache line that is accessed by a transaction holds exclusive write access. Specifically, if a cache line is speculatively accessed, it cannot exist in any non-speculative state, meaning that write ownership is provided to at most one transaction, and no non-speculative writes can occur without forcing an abort for all transactions in the system. Additionally, the commit protocol (Chapter 4.5) ensures that all writes by a transaction appear atomically across the system.

In addition, again similar to [48], HMTX instructions all include implicit memory barriers. This simplifies the programming model and ensures that all stores within a transaction are seen as a single store that is ordered with respect to its surrounding code. Because transactions tend to be very large and are in the evaluated benchmarks, the overhead here is trivial.

Chapter 8

Evaluation

8.1 Methodology

The HMTX system is modeled and evaluated using the gem5 simulator [49] in full system mode with a 4-core out-of-order processor. Table 8.1 shows the hardware configuration. Note that the cache sizes in the table are used for the base evaluation; a study on the impact of changing cache sizes on a subset of benchmarks is discussed in Chapter 8.5.

Speedup of the hottest loop of each benchmark speedup is compared. Approximately 15% of the iterations of the hot loops are measured and evaluated due to limitations of the simulation environment. Table 8.2 shows the percentage of execution time spent in the evaluated hot loop for each benchmark on a real x86 processor.

As discussed in Chapter 4.11, if a program needs to allocate memory inside an HMTX-speculatively parallelized section of the program, it cannot call into the standard library's `malloc` because it will result in misspeculation. Accordingly, all memory allocation was performed using nedmalloc [45], which uses some pre-allocated chunk of memory which is thread local. To set a fair baseline of performance comparison, versions of the benchmarks compared against (i.e. SMTX and Sequential versions) were also made to use nedmalloc.

Feature	Parameter
Architecture	Alpha 21264
Processor Cores	4
Clock Speed	2.0 GHz
L1 I and D Caches	64KB, 8-way set associative, 2 cycle latency
Shared L2 Cache	32MB, 32-way set associative, 40 cycle latency
Cache Line Size	64B
Base Cache Coherence Protocol	MOESI
Memory	1GB, 200 cycle latency
Operating System	Linux Version 2.6.27.6
Compiler	GCC Alpha Cross Compiler, Version 4.3.2

Table 8.1: Architectural Configuration in gem5.

8.2 Benchmarks

An evaluation of 8 benchmarks (7 from the SPEC benchmark suite, and 1 from MiBench) is presented, all of which need speculation for efficient TLP transformation. Of these benchmarks, 6 were also evaluated by SMTX [12]; replicated SMTX results for these 6 are directly compared against. Focus was mostly placed on those benchmarks that use the DSWP execution paradigm, as they require MTX support. The same parallelization paradigm was used for both the SMTX and HMTX versions. Table 8.2 shows the benchmarks and their parallelization paradigms, as well as the percentage of the execution time the hot loop runs for on a native x86 machine.

The benchmarks were speculatively parallelized manually for both the SMTX and HMTX versions. However, even though the HMTX versions were manually transformed, all loads and stores inside a transaction were added to the read and write sets, meaning speculation validation is performed for **every memory access** inside a transaction. This is the maximum amount of speculation validation possible for speculative parallel execution. Therefore, this represents the **worst possible case** for validation overhead, regardless of automatic or manual parallelization.

Benchmark	Parallel Paradigm	Hot Loop Native Exec Time %	Avg Number of Spec Mem Accesses Per TX	Number of TX Aborts Avoided via SLA Per TX	% of Spec Loads Needing SLA	% of Branch Insts Inside Hot Loop	Branch Mispred Rate Inside Hot Loop
052.alvinn	DOALL	85.5%	2,713,122	0.1	1.36%	11.5%	0.247%
130.li	PS-DSWP	100%	181,640,675	23.7	4.21%	20.5%	3.65%
164.gzip	PS-DSWP	98.4%	6,247,862	3.82	7.08%	14.6%	2.69%
186.crafty	PS-DSWP	99.5%	4,393,793	1.79	5.01%	13.1%	5.59%
197.parser	PS-DSWP	100%	24,727,941	22.8	2.56%	19.2%	1.05%
256.bzip2	PS-DSWP	98.5%	131,247,202	22.3	6.04%	12.6%	1.33%
456.hmmer	PS-DSWP	100%	1,707,332	0.109	1.74%	4.83%	1.03%
ispell	PS-DSWP	86.5%	43,695	0.01040	13.1%	16.6%	2.8%

Table 8.2: Statistics from simulated speculative execution using HMTX, and from native sequential non-speculative execution.

Meanwhile, the SMTX versions retained the advantage of negligible speculation validation thanks to expert transformation, with minimal read and write sets. As noted, this is not a reasonable expectation for automatic parallelization or non-expert programmers (Chapter 2.2). Even with this large advantage, HMTX compares favorably to SMTX.

8.3 Hot Loop Speedup

As seen in Figure 8.1, the HMTX system with 4 cores provides a geomean speedup of 2.04x over sequential execution on all 8 benchmarks. On the 6 benchmarks evaluated by both HMTX and SMTX, HMTX has a speedup of 2.08x, outperforming SMTX with a speedup of 1.58x.

HMTX achieves better performance than SMTX even though it performs significantly more speculation validation. One reason for the HMTX speedup is that SMTX requires an extra commit process for processing speculative memory operations (Chapter 2.3), taking up one core's resources.

Even if SMTX did not require this extra commit process, the performance comparison is not apples-to-apples. As previously noted in Chapter 2.3, because of expert manual transformation, the SMTX versions of the benchmarks perform very little alias speculation validation. Meanwhile, benchmarks using the HMTX system performs speculation validation for *every memory access* inside a transaction.

Recall that in Figure 2.2, when programs parallelized with SMTX perform more speculation validation (though still less than the HMTX versions), their performance degrades badly. Thus, these systems do not provide a realistic path forward for automatic parallelization. Meanwhile, HMTX achieves good performance with the largest possible read and write set. Thus, even with alias analysis that cannot disprove many potential dependences between threads, good performance could still be achieved with more realistic expectations of a compiler. Accordingly, the HMTX system could provide a solid building block toward

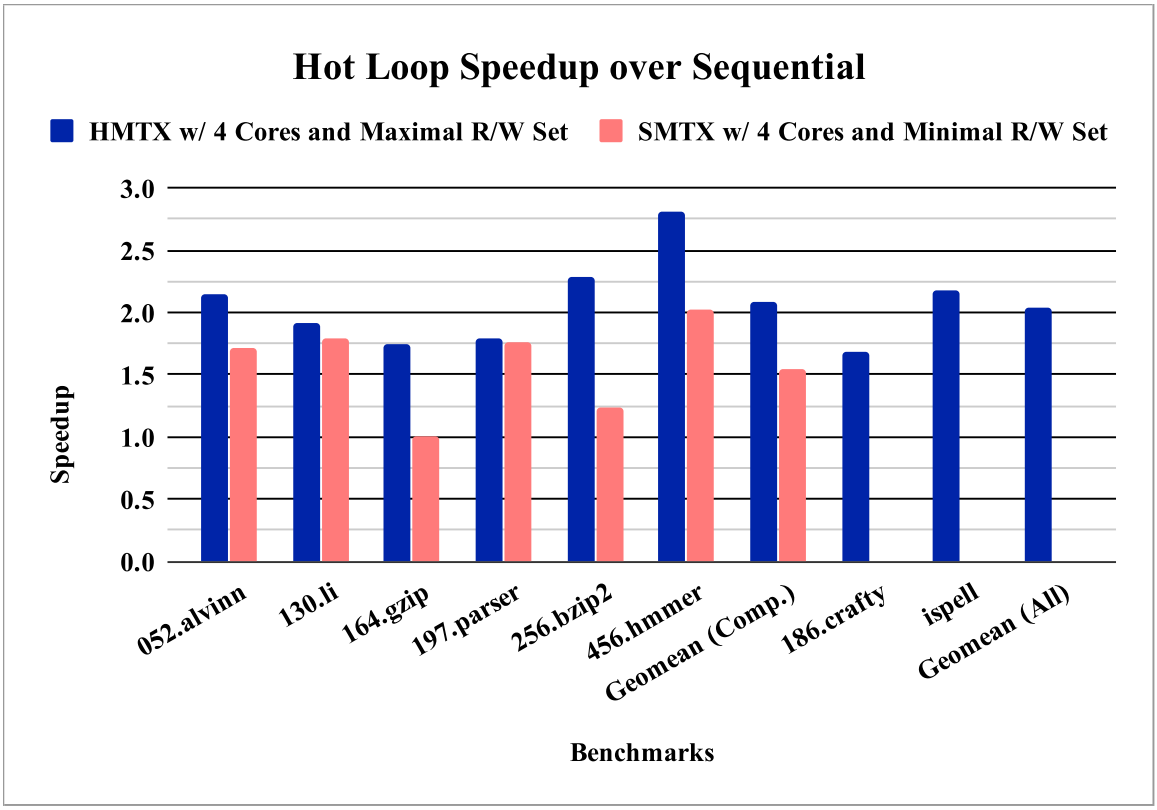


Figure 8.1: Hot loop speedup over sequential using 4 cores. SMTX versions have minimal read and write sets due to expert manual transformation. HMTX versions perform speculation validation on *every read and write* inside a transaction, i.e. the maximum possible read and write set. Note that there is no SMTX comparison for 186.crafty and ispell; accordingly, "Comp." represents those benchmarks with an SMTX comparison, while "All" represents all benchmarks.

enabling automatic parallelization.

8.4 Aborted Transactions

As discussed in Chapter 4.9, there are two potential sources of aborts in HMTX: those due to invalid speculative assumptions (such as control or alias misspeculation) (Chapter 8.4.1), and those aborts due to a lack of capacity in the cache for tracking speculative state (Chapter 8.4.2). Neither kind of abort occurred in the benchmarks that were evaluated.

8.4.1 Aborts Due to Incorrect Speculative Assumptions

Potential sources of aborts such as control or alias misspeculation were not experienced, as only high confidence speculative parallelization is performed. This is the same as what was found by the SMTX system, which was evaluated on 6 of the same 8 benchmarks using similar parallelization schemes. Aborts are costly regardless of the underlying TM system, and so are generally avoided.

In addition, thanks to speculative load acknowledgments (SLA) (Chapter 5.1), all benchmarks avoided false misspeculation due to branch misprediction. Recall that this is a possibility if an already-dispatched HMTX-speculative load is squashed due to branch misprediction, and the load incorrectly marks the cache line as speculatively accessed by some transaction. An SLA prevents marking such lines as speculative and thus avoids false misspeculation.

The number of avoided misspeculations, seen in Table 8.2, varies for each benchmark depending upon the data access patterns given their complex data structures and control flow. Table 8.2 shows these number of misspeculations that were avoided thanks to SLAs, as well as the branch misprediction rate and the percentage of instructions that are branches.

In general, the higher the branch misprediction rate and percentage of branch instructions, the higher the number of avoided aborts. For example, 052.alvinn and 456.hmmer have low misspeculation rates and low rate of branches overall, and both require less SLAs and avoided less false misspeculations per transaction. For example, 130.li makes large use of function pointers and thus indirect jumps, leading to a high branch misprediction rate (3.65%). Combined with its high rate of branches (20.5%), 130.li has a very high number of false misspeculations avoided.

Table 8.2 additionally shows the number of SLAs that are sent as a percentage of the number of speculative loads performed by the system. Thanks to memory locality, most speculative accesses are to lines that have already been marked as speculative with that specific VID. Thus, there is not a significant amount of extra requests sent to the caches,

and there is minimal impact on performance.

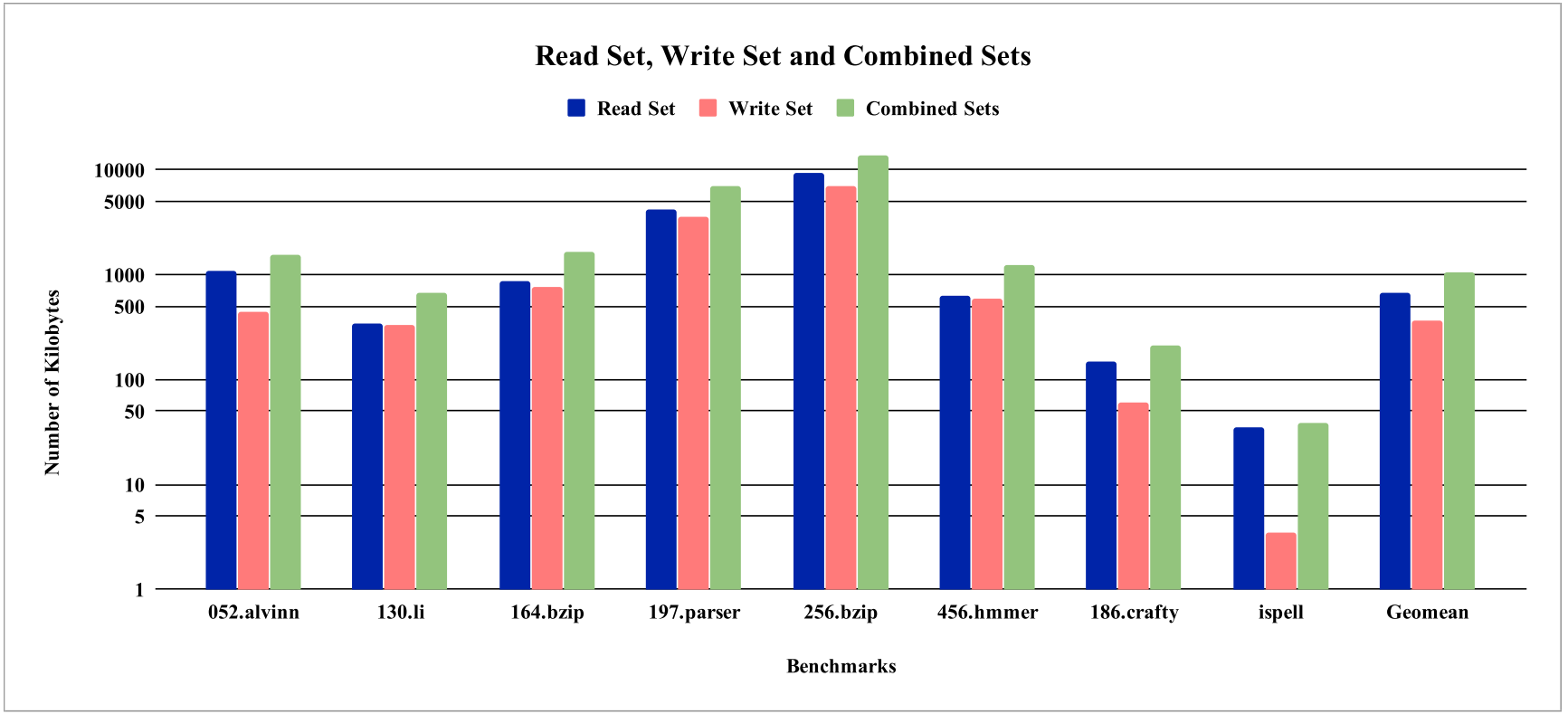
8.4.2 Aborts Due to Lack of Capacity for Speculative Memory

An abort could also be triggered if certain speculatively modified lines overflow the caches (Chapter 4.9). However, this was not seen in the evaluated benchmarks. Only 197.parser and 256.bzip2 had the allowed non-speculative versions of speculatively read lines overflow the caches, which does not cause an abort (Chapter 5.4).

Figure 8.2 shows the average size of the read and write sets. Clearly there is wide variation of the size of each transaction depending on the benchmark. The largest transaction sizes belong to 256.bzip2. However, note that this does not perfectly correspond to the number of speculative memory accesses as seen in Table 8.2; 256.bzip2 has the second largest number of speculative memory accesses, while 130.li has the first even though it is below average in transaction sizes. This is due to memory locality; 130.li tends to access the same speculatively accessed memory repeatedly, while 256.bzip2 has less locality, spreading its accesses more throughout memory.

8.5 Study of Performance vs. Cache Sizes

As previously mentioned in Chapter 4.4 and Chapter 5.4, many speculative copies of lines are kept across the cache system, and most cannot overflow the caches without forcing an abort. This means that there is greater cache pressure as the size of transactions becomes very large. Given that the initial evaluation is done using relatively large cache sizes, a study was performed on two benchmarks, varying the cache sizes in order to gauge how cache sizes impact it.



Benchmark	052.alvinn	130.li	164.bzip	197.parser	256.bzip	456.hmmer	186.crafty	ispell	Geomean
Read Set (kB)	1085	342	877	4175	9302	636	148	35	673
Write Set (kB)	444	331	762	3562	6921	596	61	3	370
Combined Sets (kB)	1529	673	1639	7003	13904	1232	209	39	1064

Figure 8.2: (Top) Bar chart depicting average size of the read and write sets in kilobytes. (Bottom) Table displaying the raw data.

This study was performed on 130.li and 256.bzip2. 130.li was about average for transaction size and thus speculative memory usage, while 256.bzip2 was the largest. Thus, they appear to be good candidates for performing this evaluation.

Note that as the cache sizes shrink, there is less space for speculative memory. Thus, it's natural that at some point speculative memory that must stay in the caches to avoid an abort will in fact overflow the caches and would normally force an abort to occur.

In order to perform a study on performance in spite of this, such speculative memory was allowed to overflow caches and execution was allowed to proceed. This was accomplished by implementing versioned memory that saves the speculative state of cache lines. Thus, the evaluation assumes that a speculative version of a line is able to be stored and retrieved from memory using normal memory latency. Such configurations are marked with an asterisk on their configuration. This occurred for many of the bzip2 configurations, and one 130.li configuration. In the figures, configurations with an asterisk represent that this overflow into memory occurred.

Across these experiments, some cache configuration consisting of L1 and L2 cache sizes was selected, and then run on the sequential, SMTX, and HMTX versions of the benchmark.

Note that the HMTX system uses extra cache lines to store multiple versions of lines, representing different access patterns by the many transactions which may be reading and writing them. Thus, there is more cache pressure. As caches sizes shrink it is expected that the impact of this extra cache pressure will be comparatively worse speedups (Chapter 8.5.1) and worse cache hit rates (Chapter 8.5.2).

8.5.1 Speedups Across Varying Cache Configurations

Speedup in this section is calculated by comparing the sequential performance on a specific cache configuration with the SMTX and HMTX performance on that same cache configuration.

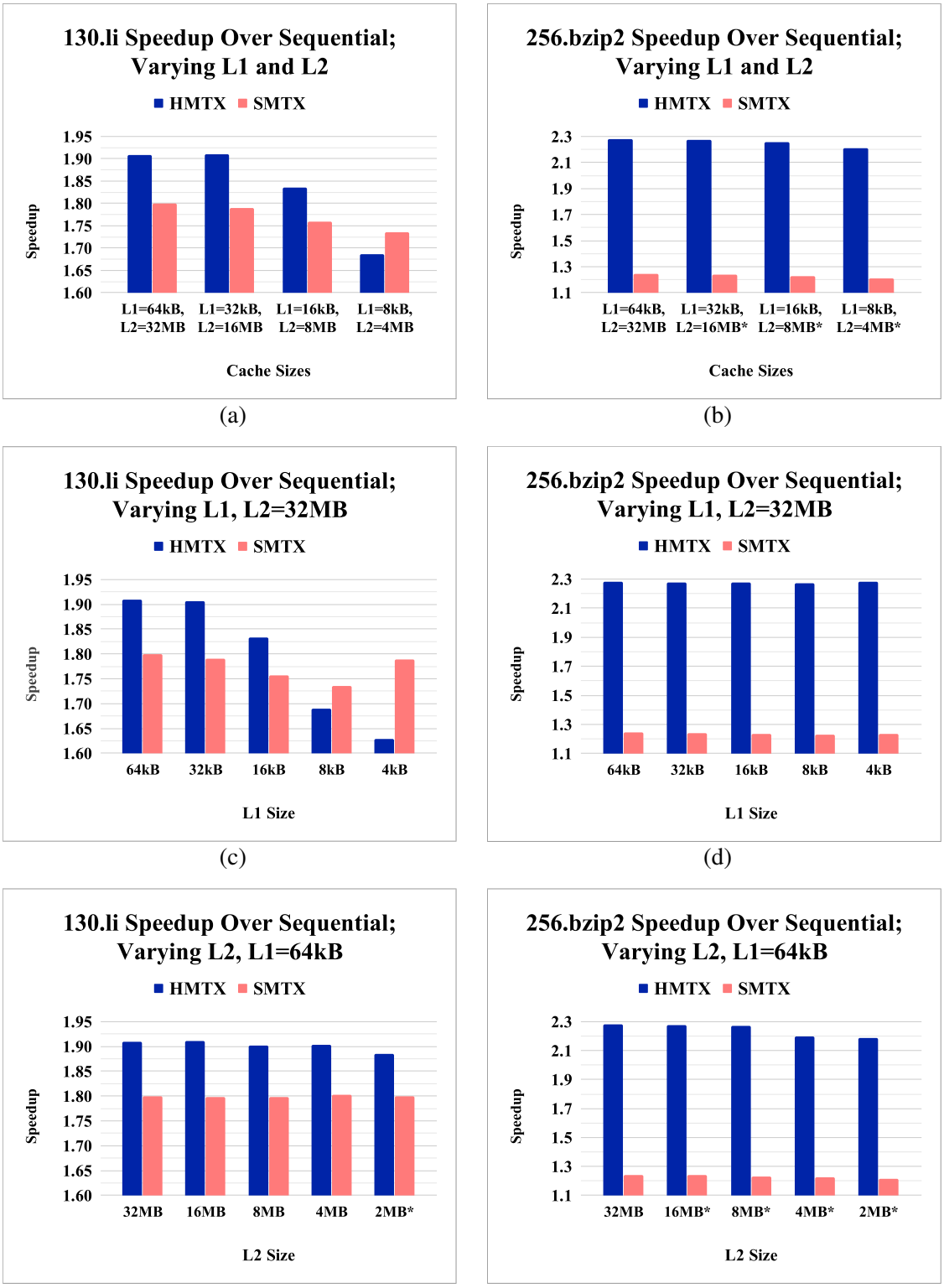


Figure 8.3: HMTX and SMTX speedup over sequential execution for 130.li ((a), (c), (e)) and 256.bzip2 ((b), (d), (f)) over various cache configurations. Across the x-axis, one or both of the L1 and L2 are halved for each configuration. If a configuration has an asterisk, a dirty speculative memory overflowed the L2 cache.

First, an experiment is show where both L1 and L2 cache sizes were halved repeatedly to examine performance degradation. As seen in Figure 8.3a, performance of 130.li clearly degraded as the cache sizes shrink; speedup degradation appears to accelerate as the caches are shrunk, from a speedup of 1.91x in the original cache configuration of {`L1=64kB, L2=32MB`}, down to 1.69x once the caches are dropped all the way to {`L1=8kB, L2=4MB`}. Note that these cache sizes are much smaller than some mobile processors; the 2017 Apple A11 found in iPhones had larger caches, with L1=64kB and L2=8MB [50].

Additionally, Figure 8.3b shows the performance of 256.bzip2. Here the performance of the benchmark varies much less as the cache sizes are halved. Taking the L1 cache size down to 8kB and the L2 cache size down to 4MB showed only a marginal decrease in speedup from 2.28x to 2.21x, a slight 3% difference. However also note that from the second configuration on, the L2 cache is already overflowing with speculatively accessed lines which force abort.

In order to determine the cause of this performance degradation, additional experiments were run holding the L2 cache size constant while varying the L1 cache size, and vice versa. Figure 8.3c shows speedups on 130.li as the L1 cache size is shrunk from 64kB down to 4kB, holding the L2 cache size constant at 32MB. As seen in the figure, HMTX 130.li is very sensitive to the L1 cache size. Meanwhile SMTX seems to slowdown initially but sees almost no overall speedup loss on the smallest L1 cache size.

Figure 8.3e shows that shrinking the L2 size had minimal impact on speedup; HMTX had a slight decrease in speedup, while SMTX showed no difference. Clearly the performance loss in HMTX seen in Figure 8.3a was mostly due to the L1 cache size shrinking.

These same experiments were run on 256.bzip2, varying either only the L1 or L2. 256.bzip shows different sensitivity to shrinking cache sizes compared to 130.li. As seen in Figure 8.3d, L1 cache sizes seem to have no impact on 256.bzip2 speedup over sequential execution. The speedup degradation only comes into play when decreasing the size of

the L2 cache, as seen in Figure 8.3f. Essentially the opposite is seen here versus 130.li. Again note that from the second configuration on, the L2 cache is already overflowing with speculatively accessed lines which force abort.

8.5.2 Cache Miss Rates Across Varying Cache Configurations

The speedup degradation over sequential execution as cache sizes vary as shown in the previous section is expected. To look deeper into the impact of the shrinking cache sizes and its impact on performance, the L1 and L2 cache miss rates are displayed across these same cache configurations for HMTX, SMTX, and Sequential execution in Figure 8.4.

Note that this is not an apples-to-apples comparison; while Sequential execution is single threaded, HMTX and SMTX use multiple worker threads and processes, respectively. Taking an average of all the threads for HMTX and SMTX would not be representative of the impact on speedup, because there are threads off of the critical path which do not impact speedup.

Instead of averaging all of the threads' miss rates, the miss rate shown is an average of the threads which are on the critical path, i.e. those threads which are part of the parallel stage for these two benchmarks. For Sequential, there is only one thread and critical path, and so its miss rate is across the whole loop.

Figure 8.4a shows the miss rates for 130.li on the L1 while it varies in size. As expected, the L1 has a higher miss rate than the SMTX and Sequential versions of the benchmark. As the L1 shrinks, all L1 miss rates increase. However, the gap between HMTX and both SMTX and Sequential grows as the L1 shrinks, for example compared to SMTX from a difference of 2.1% at 64kB to 3.0% at 4kB.

Examining 256.zip2, it has somewhat similar L1 miss rate characteristics in Figure 8.4b. All configurations see their miss rate increase as the L1 cache sizes shrink, and similar to 130.li the gap between HMTX and its counterparts grows, for example compared to SMTX from a difference of 0.18% at 64kB to 0.27% at 4kB.

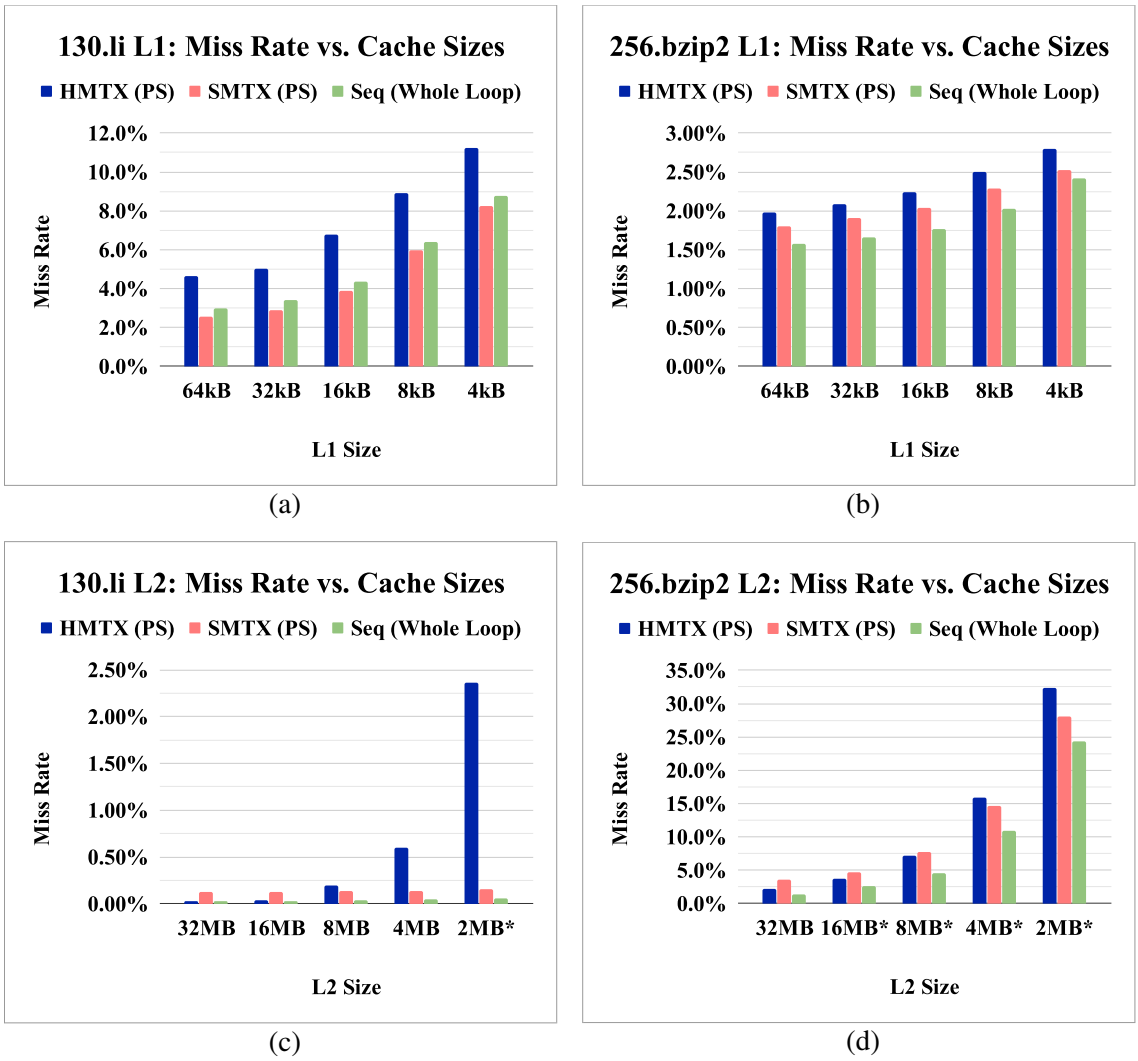


Figure 8.4: Cache miss rates for 130.li ((a) and (c)) and 256.bzip2 ((b) and (d)) across various cache configurations. Cache sizes are halved across the x-axis, with the same configurations as from Figure 8.3. For HMTX and SMTX, the miss rate shown is an average of the threads which are on the critical path, i.e. those threads which are part of the parallel stage (PS) for these benchmarks. For Sequential, there is only one thread, and so its miss rate is across the whole loop, meaning it also includes the logic from other stages not measured for HMTX and SMTX. If a configuration has an asterisk, a dirty speculative memory overflowed the L2 cache.

Additionally, Figure 8.4c shows the miss rates for 130.li on the L2 while it varies in size. Again, miss rates increase as the L2 shrinks. However, the L2 miss rate increases significantly for HMTX from 0.02% at 32MB to 2.36% at 2MB, while miss rates stay very low for both SMTX and Sequential paradigms (though still growing slightly).

For 256.bzip2 in Figure 8.4d, all configurations see a large increase in L2 miss rates as the L2 shrinks. Note though that HMTX's miss rate increases at a faster rate.

As seen in both Figure 8.4c and Figure 8.4d, with the initial cache configuration, HMTX has a better L2 hit rate than SMTX. One reason for this is that SMTX relies on copy-on-write semantics at the page level for keeping versions of memory separate. This means that there is potential for a lot of unnecessary memory duplication; for example if during a transaction only a single byte on a page is written to, the entire 4kB page is duplicated, whereas in HMTX only that cache line would have a new version created.

Another reason for this is that SMTX requires an extra commit process for managing speculative state. This means there is an entire extra copy of all pages with speculative memory modifications that exists.

These two factors combined mean that there could be a lot of extra memory pressure in SMTX not seen in HMTX, depending on the access patterns of the benchmark. Still, such extra memory can spill over the L2 to memory in SMTX, while in HMTX it must fit in caches. Thus as the L2 shrinks there is less space for non-speculative memory to fit and miss rates increase at a faster rate for HMTX.

8.5.3 Area, Power, and Energy

Area and power are modeled with McPAT [51]. The 22nm technology node is used. Power gating and low L2 cache standby power are utilized. Table 8.3 displays statistics gathered.

An HMTX system with 4 cores has a total area of 111.1 mm^2, 4.0 mm^2 larger than the base system with the same cache sizes and core count (107.1 mm^2), which was used for SMTX evaluation. The largest source of these increases in the HMTX system is adding 12

bits to every line in the cache, 6 each for the modVID and highVID, as well as the low-high cascading comparators as discussed in Chapter 4.6.

McPAT uses CACTI [52] in order to model caches, which performs architectural modeling of SRAM based caches. Total leakage increases marginally when adding in HMTX extensions (Table 8.3). Additionally, geomean runtime dynamic power consumption increases for HMTX due to the aforementioned logic and cache modifications for HMTX. Overall, energy consumption with HMTX is lesser than for SMTX, largely due to the difference in execution time.

Applications running on hardware with HMTX extensions would still have an increase in energy consumption even if they do not utilize HMTX functionality. To evaluate this impact, the same SMTX and sequential benchmarks were run on HMTX hardware through McPAT. Note that this has no impact on execution time. Overall, geomean runtime dynamic power and energy consumption increased marginally (Table 8.3). This highlights the low impact of HMTX extensions.

Hardware	Exec Model	Area (mm^2)	Total Leakage (W)	GM Runtime Dynamic (W)	GM Energy (J)
Commodity	Sequential (All)	107.1	5.515	3.578	7.322
	Sequential (Comp.)			3.655	10.913
	SMTX, Min R/W			13.733	14.846
Commodity + HMTX Extensions	Sequential (All)	111.1	5.607	3.619	7.430
	Sequential (Comp.)			3.696	11.073
	SMTX, Min R/W			13.945	15.081
	HMTX, Max R/W (All)			14.617	7.956
	HMTX, Max R/W (Comp.)			14.723	11.553

Table 8.3: Area, power, and energy results on a simulated 4-core machine. "All" represents all evaluated benchmarks, while "Comp." represents only those benchmarks with an equivalent SMTX version to compare against. Note the difference in geomean (GM) energy between "Comp." and "All" is largely due to the short execution time of ispell compared to other benchmarks.

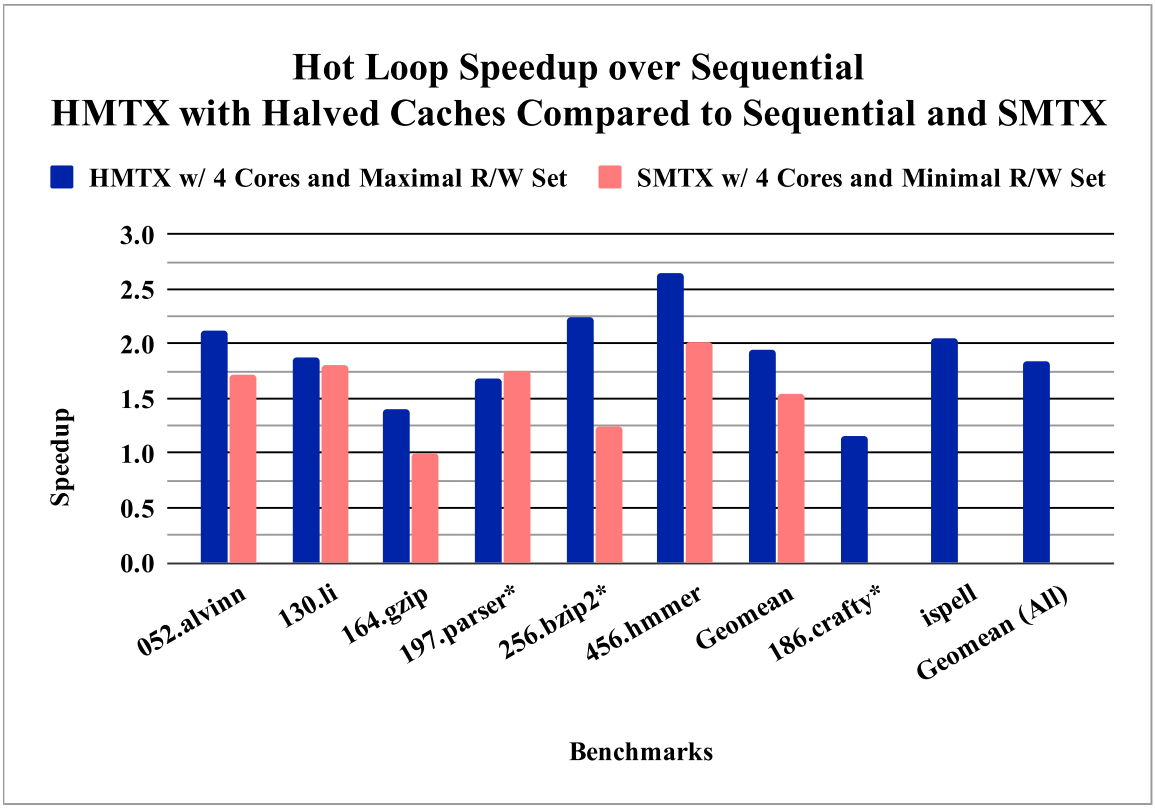


Figure 8.5: Hot loop speedup over sequential using 4 cores. SMTX and Sequential versions use an L1 of 64kB and an L2 of 32MB, while HMTX versions use an L1 of 32kB and L2 of 16MB. SMTX versions have minimal read and write sets due to expert manual transformation. HMTX versions perform speculation validation on *every read and write* inside a transaction, i.e. the maximum possible read and write set. If a benchmark has an asterisk, for the HMTX version dirty speculative memory overflowed the L2 cache.

8.6 Cost Benefit Analysis of HMTX Extensions

HMTX extensions require additional processor area and complexity. Instead of adding the requisite logic for HMTX, processor architects could instead opt to use that extra area and complexity for other purposes, such as larger cache sizes. This section provides some context and quantitative evidence that the performance gain from larger cache sizes does not provide close to the same speedup potential as HMTX provides, and thus that the benefits of adding HMTX extensions outweigh the benefits gained by another potential use of processor area.

Figure 8.5 shows the results of an experiment where benchmarks parallelized using

	Sequential and SMTX; L1=64kB, L2=32MB	**HMTX; L1=32kB, L2=16MB**
Area (mm^2)	107.1	82.59
Leakage (W)	5.515	5.477

Table 8.4: Area and Leakage for both HMTX with L1 of 32kB and L2 of 16MB, and Sequential and SMTX with L1 of 64kB and L2 of 32MB.

HMTX were run with half the cache sizes of the sequential and SMTX versions. This illustrates that a smaller, less complex system that includes HMTX extensions outperforms a more larger, more complex system without HMTX extensions thanks to the low-overhead transactional memory support HMTX provides. Thus, the opportunity cost of choosing greater cache sizes instead of HMTX extensions is quite high.

The geomean speedup on all benchmarks using HMTX is still significant at 1.84x. Additionally the comparison benchmarks for both HMTX and SMTX still show that the geomean speedup using HMTX (1.95x) is higher than SMTX (1.55x). Note that 197.parser with HMTX provides a speedup of 1.69x, which is slightly slower than the speedup using SMTX of 1.76x.

Additionally, again recall that SMTX uses a minimal read and write set, while HMTX uses a maximal one, so although the gap between the two becomes closer, HMTX is still performing significantly more transactional coverage. Also note that as discussed in Chapter 8.5, 197.parser, 256.bzip2, and 186.crafty using HMTX overflows the L2 cache with speculative memory. This means these benchmarks would not be able to use HMTX extensions; however, future work (Chapter 10) discusses ways to allow for these benchmarks to use HMTX with a processor with such smaller sized caches.

Lastly, Table 8.4 shows a comparison of area and leakage for HMTX with halved caches versus the base configuration used by Sequential and SMTX. HMTX uses significantly less area, as it is 77% the size of the base configuration without HMTX extensions. Additionally, leakage is marginally less as well.

Chapter 9

Related Work

9.1 MTX by Vachharajani [1]

The HMTX system follows Vachharajani's lead by adding version IDs to each cache line, as noted in Chapter 2.3. However, there are some important differences between the two works.

Speculative Memory Processing Efficiency. Vachharajani's commit protocol is prohibitively expensive, both in complexity and time. On commit, the entire cache must be searched for every line with the committed VID (similar to the naïve version in Chapter 4.5). Even with an ORB-like structure [15] that holds the address of every speculatively accessed line, processing every speculative line individually on every commit would still be very slow. Additionally, the protocol requires broadcasting an invalidation for each speculatively modified line to gain exclusive access to it. This would lead to considerable bus contention and further degrade performance. Lastly, the abort implementation is not discussed in detail, and VID overflow is not considered.

In contrast, HMTX is designed so that the state of other versions of the same line does not need to be known, nor does there need to be an invalidation or interaction with them to perform a commit (Chapter 4.5). This allows for commits to occur lazily, similar to other

works [34, 47]. This simpler, lazy approach is not bursty or time consuming in searching an entire cache or processing all lines at once, allowing for transactions that speculatively access large amounts of data to commit quickly and efficiently.

Cache Pressure. Vachharajani's work creates a new version of a cache line for every read from a new version. This may lead to unnecessary cache pressure as many read-only lines redundantly store the same data. In contrast, HMTX only creates new lines when a speculative write occurs to a line that has not yet been speculatively written for the given transaction's VID. In addition to reducing cache pressure, this also allows for transactions with larger read and write sets.

Commit Granularity. Vachharajani's byte-level commit granularity requires much higher space and complexity. In contrast, HMTX uses cache line-level granularity. This reduces the complexity of implementation at the cost of potential false misspeculation; however, the evaluated benchmarks did not experience any such false misspeculation due to only performing high-confidence speculative parallelization.

Evaluation. Vachharajani's work only provided a description of its design. This design was not modeled or evaluated in any way. In contrast HMTX presented in this dissertation was modeled in the gem5 simulator using full system mode, meaning evaluated benchmarks run on top of a real Linux kernel, providing a real-world environment to evaluate on.

9.2 SMTX and DSMTX

SMTX [12] and DSMTX [13] are software-only MTX TM systems. They are able to achieve good results on commodity systems. Both of these systems use a "privatized by default" memory model, as discussed in Chapter 4.10. To accomplish this, SMTX and DSMTX use Copy-On-Write (COW) and Copy-On-Access (COA) semantics, respectively, at the page granularity to manage multiple versions of memory. SMTX's COW semantics

mean it will duplicate pages when one of the worker threads attempts to write to a page that it has not yet written to. Meanwhile DSMTX's COA does the same but on first access instead of write.

Because these systems create copies of memory at the page boundary, this can result in significant unnecessary memory duplication, which can hurt cache hit rates as discussed in Chapter 8.5.2. In contrast, HMTX uses a "shared by default" approach to managing speculative versions of memory, meaning all memory is in a single memory space, and if privatization is necessary it must be done explicitly. Because HMTX creates multiple versions of memory at the cache line granularity, less memory needs to be duplicated.

Additionally, both SMTX and DSMTX use software queues to communicate all speculative reads and writes between each other and to separate processes that manage validating speculative accesses and committed, non-speculative program state. Note that benchmarks for these systems are manually transformed and thus are expertly tuned with minimal read and write sets to avoid the validation overhead that would be required during automatic parallelization, or by transformation by a non-expert. As shown in Figure 2.2, performance heavily depends upon read and write set sizes. Meanwhile, HMTX uses a single memory space and relies on the modified cache coherence protocol combined with VIDs on each access and all cache lines to implicitly use the correct version of memory in a low-overhead manner.

9.3 Single-Threaded TM Systems

No past hardware TM systems have sufficient support for multi-threaded transactions via both uncommitted value forwarding and group transaction commit. Consequently, these systems cannot support speculative pipeline parallel execution. By contrast, HMTX can support a wide range of speculative execution paradigms, from TLS to DSWP-style execution.

Similar to HMTX, versioned memory is used by some TM systems to manage transactions [15, 17, 33, 34, 47, 53, 54]. This enables lazy commits and holding speculative state from multiple tasks in a single cache, which are both used by HMTX. However, none of these systems allow for a single transaction's speculative memory to migrate to other peer caches, which is a requirement for pipeline parallelization (Chapter 2.3).

Many past systems provide an ordering for transactions as HMTX does, allowing for uncommitted value forwarding as an optimization [15, 28, 33, 35, 36, 55, 56, 57]. However, as noted in Chapter 2.3, group commit is also required in order to ensure that all speculative modifications from a single transaction, likely spread across multiple caches, are atomically committed.

Additionally, while some TM systems support large read and write sets [47, 58, 59, 60], most cannot support transactions as large as those in the parallelized benchmarks presented. Thus, even if they did support uncommitted value forwarding and group transaction commit, they would be unable to perform speculative pipeline parallelization.

HMTX utilizes a transactional state that is similar to the "executed but not committed" described in the STM presented in [61]. However, while this is a performance optimization in that work, it is a requirement for collaboration on a single transaction with pipelined parallelism using HMTX, as it allows a thread to move on to executing its next part of a transaction as part of a pipeline, expecting future threads to complete and commit the transaction.

Lastly, all prior systems are susceptible to false misspeculations due to branch misprediction, which HMTX overcomes via SLAs (Chapter 5.1).

TM System	Multithreaded Transaction Support	Little to no OS Support Required	No Runtime System/ Meta Threads	Scalable	Inter-TX Uncommitted Value Forwarding	Granularity
HMTX [22] (this work)	✓	✓	✓	✓	✓	Line
MTX by Vachharajani [1]	✓	✓	✓	✗	✗	Byte
SMTX [12]	✓	✗	✗	✓	✗	Byte
DSMTX [13]	✓	✗	✗	✓	✗	Byte
TCC [18]	✗	✓	✓	✗	✗	Line
Hydra [28]	✗	✓	✓	✗	✓	Byte
SVC [34]	✗	✓	✓	✗	✓	Byte
MDT [17]	✗	✓	✓	✓	✓	Word
Scalable TLS [15]	✗	✓	✓	✓	✗	Byte
Herlihy and Moss [62]	✗	✓	✓	✗	✗	Line
Intel Haswell TM [63]	✗	✓	✓	✓	✗	Line

Table 9.1: Comparison of HMTX to other works.

Chapter 10

Future Work

10.1 Automatic Parallelization

A large motivation of this work is to take a big step closer to automatic parallelization. The hope is that a compiler could achieve profitable automatic speculative parallelization with the help of low overhead speculation validation via HMTX.

Significant prior work in automatically parallelizing compilers has been done by many researchers, including those of the Liberty Research Group at Princeton University. As mentioned in Chapter 2.3, Johnson [20] found that in such automatic compilers [30, 31], "imprecise analysis forces the compiler to compensate with more speculation . . . Increased validation overheads cause application slowdown."

If HMTX was integrated into such an automatically parallelizing compiler, and potentially combined with other techniques such as privatization mentioned in Chapter 4.10 and auto-tuning mentioned in Chapter 10.2, automatic parallelization would likely take a big leap forward.

10.2 Automatic Tuning to Support Optimal Parallelism and Performance

As discussed in Chapter 2.1, there are multiple parallel paradigms to choose from when parallelizing a loop. Once one paradigm is chosen, we also need to then choose the number of threads used to execute this loop with this paradigm.

In general, we ideally would attempt to use all parallel resources available when parallelizing a program. That could be with DOALL-style parallelism in which all parallel resources perform the same task, or PS-DSWP-style parallelism in which there are parallel DOALL stages, or even DOACROSS style-parallelism in which we could attempt to use more threads to execute each iteration of a loop.

However, ramping up the number of threads means there is more speculative memory in caches. Thus, the more speculative parallelism that is employed, the higher the chance that an abort due to lack of cache capacity (Chapter 4.9) will be forced. Thus, there is a balance here we want to strike: increase the number of parallel speculative threads executing without increasing parallelism so much as to force a capacity abort.

In this dissertation, benchmarks have been configured to execute given a specific number of threads. This number does not deviate during execution. Instead, a system such as Parcae [64] would be well-suited to take on such a task. It could be made more aware of speculative TM-based execution schemes, in addition to information about previous attempts to increase the number of parallel threads for a program. It could then use this information to tune the number of threads at runtime, increasing the number of threads only when it is confident that capacity aborts will not occur, which would degrade program performance.

10.3 "Shared by Default" vs. "Privatized By Default"

As mentioned in Chapter 4.10, the SMTX and DSMTX systems spawn parallel workers through process forking. This means that all memory is automatically "privatized by default" via process separation, and any accesses that need speculative verification must be explicitly communicated to the commit process. In contrast, HMTX uses "shared by default", wherein all memory is shared between worker threads, and any private memory for threads must be explicitly privatized.

Future work could attempt to use a hybrid of these two models, instead of privatizing everything by default like SMTX or nothing by default like HMTX. Memory locations that need speculation validation or that communicate dependences between threads would be allocated in shared memory between the processes and registered with the HMTX system. All other memory that should be privatized would use its own address space. This could allow for the speculative pressure to be much lower in HMTX for some programs, potentially enabling more workloads, allow for smaller caches to support the presented workloads, or both.

A system such as Privateer [44] could be used with HMTX to automatically privatize memory when needed, perhaps in conjunction with Parcae [64] mentioned in Chapter 10.2.

10.4 Speculative Memory Leaving the Cache

In the presented implementation, all caches in the system have MTX support. If certain speculative lines are selected as victims to be written back to main memory, the speculative information of the lines (metadata about transactions which accessed the line, speculatively modified data of the line, or both) would be lost when written back. Therefore, transactions greater than or equal to the lowest VID in the line would need to abort.

In order to prevent this required abort from occurring, a structure could be added on the path to main memory, similar to [47]. When the last level cache picks a speculative

line as a victim, the structure could store information about this line. If the line was not speculatively modified then the structure saves the address, VIDs, and coherent state of the line. If the line was speculatively modified then the line's data is also stored in the structure. In practice, the clean case is much more common than the dirty case.

When a new access comes in for this line to main memory, the line can be correctly sent back with the VIDs and coherent state it had before it was ejected. On commit, the only required action is updating the latest committed VID. On abort, the structure must transition lines according to the abort state diagram described in Chapter 4.5.

This structure is only necessary for applications which have very large sets of speculatively accessed lines, such that speculative memory overflows caches (Chapter 8.4.2).

Alternatively, schemes based on logging [65] and/or probabilistic methods such as Bloom filters [16, 66, 56] could be used to track speculatively accessed lines that overflow the cache. This could also be done via paging or other software mechanisms [58, 59, 60].

10.5 Scaling to More Cores and Bigger Workloads

Consumer grade systems continue to be underutilized; past work shows that even 4 cores appears to be over-provisioning for most applications [8, 9, 10]. Still, future work could explore performance across more cores. This could be done by adapting the HMTX coherence scheme to a directory-based protocol to allow for more efficient scaling. To deal with overflowing memory from scaling these systems up(Chapter 4.9), techniques such as those discussed in Chapter 10.4 could be utilized.

10.6 Better Software Support for Needed Speculative Operations

As discussed in Chapter 4.11, there are limitations on programs that must be worked around before the program can be speculatively parallelized with HMTX.

One such limitation is on calls into the kernel. All such calls must be non-speculative. This includes I/O operations; inputs must be read without side effects, and outputs must be buffered until commit time. HMTX benchmarks explicitly read inputs non-speculatively and buffered outputs until committed. Prior work [30] created a transactional I/O system to overcome this instead, which could be used if adapted to work with HMTX. Alternatively this sort of support could be integrated more directly into the OS.

This also includes memory allocation through the operating system. This is because the system calls such as `sys_brk` in Linux that are used to implement `malloc`, `free`, `mmap`, etc. will end up interacting with each other and causing spurious misspeculations. Better support could be integrated into the OS for memory allocation that is HMTX aware.

10.7 Using HMTX With Programs That Are Already Multithreaded

Much of the focus of this dissertation is on parallelizing benchmarks which are originally single-threaded, sequential programs. However, HMTX can also be used with programs that are already made up of multiple threads.

One common use of transactions is for multiple threads executing in a DOALL manner, for example in database applications where multiple threads perform atomic operations on a database. HMTX could be used by such programs; assigning each transaction a VID works nicely in this model, and allows for uncommitted value forwarding in the same way that past systems have, by assigning the transactions some order and allowing younger

transactions to forward uncommitted data to older transactions.

Another way a multithreaded program could use HMTX is for each thread to use HMTX to speculatively parallelize multiple loops. For example, if two threads have their own hot loops that they would like to parallelize using PS-DSWP, then they could do so. The downside here is that there would need to be some coordination between the two hot loops to make sure that they do not share VIDs. This means that they would also need to synchronize when they jointly used up all VIDs in a flight. Additionally, if one of the loops had misspeculation then both would need to fall back, as there is no ability to differentiate between the origin of an VID. Future work could also try to address this if it is found to be problematic.

10.8 Bringing HMTX Into Real World Usage

There is much work to be done to get HMTX to be usable in real world environments, so that it is a realistic tool for programmers and compilers. Some questions that must be investigated and resolved include:

- How can languages be extended to more naturally use HMTX if a programmer would like to explicitly parallelize their program? For example, how might HMTX be used by the C++ atomics operations library, and in what ways could it be extended to e.g. make speculative pipeline parallelism more accessible?
- How can HMTX be used in conjunction with debuggers to aid in debugging speculatively parallelized programs? What should be the debugging workflow if a programmer needs to set a breakpoint inside an HMTX-speculative loop?
- How can HMTX be used by performance profilers? How can profilers be extended to make it easier for programmers to determine potential performance pitfalls when using HMTX? For example, false sharing could cause false misspeculation due to cache line granularity of VIDs, which is especially important to uncover proactively.

Chapter 11

Conclusion

Hardware Multithreaded Transactions provide a path forward for extracting pipelined thread-level parallelism from sequential programs. This dissertation details a design and evaluates an implementation of HMTX that uses very large transactions which are required when parallelizing programs that are hard to prove are thread-level parallelizable. This is often the case due to the limits of static analysis, which makes it hard or impossible to make use of profitable parallelization opportunities.

While HMTX supports speculative thread-level pipeline parallelization such as speculative PS-DSWP, an important class of parallel execution techniques, this is a superset of parallelization paradigms supported. More traditional DOALL or DOACROSS parallelization schemes are also supported. All of these paradigms benefit from techniques introduced in this dissertation, such as avoiding false misspeculations via speculative load acknowledgments (Chapter 5.1); providing for resilient transactions which can survive interrupts and exceptions (Chapter 5.2); and allowing for performant, very large transaction sizes through techniques such as lazy commit processing (Chapter 5.3).

Thus, HMTX provides MTX as well as resilient, long-running transactions without excessive hardware cost. On a multicore machine with 4 cores, a geomean speedup of 104% is achieved over sequential execution, mostly using speculative PS-DSWP, and with mod-

est increases in power and energy consumption. Future work using HMTX could utilize it alongside automatic parallelization, potentially unlocking much better performance on multicore processors with little burden to programmers who would prefer to write single-threaded programs.

Bibliography

[1] Neil Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[2] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 12th International Conference on Compiler Construction*, 2002.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[4] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[5] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 105–118, 2005.

[6] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[7] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 356–369, 2007.

[8] Cao Gao, Anthony Gutierrez, Madhav Rajan, Ronald G. Dreslinski, Trevor Mudge, and Carole-Jean Wu. A study of mobile device utilization. In *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '15, 2000.

[9] Yifan Zhang, Xudong Wang, Xuanzhe Liu, Yunxin Liu, Li Zhuang, and Feng Zhao. Towards better cpu power management on multicore smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 11:1–11:5, New York, NY, USA, 2013. ACM.

[10] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

[11] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[13] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[14] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

[15] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 1–12, June 2000.

[16] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[17] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 13–24, New York, NY, USA, 2000. ACM Press.

[18] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[19] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[20] Nick P. Johnson. *Static Dependence Analysis in an Infrastructure for Automatic Parallelization*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, September 2015.

[21] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. The influence of malloc placement on TSX hardware transactional memory. *CoRR*, abs/1504.04640, 2015.

[22] Jordan Fix, Nayana P. Nagendra, Sotiris Apostolakis, Hansen Zhang, Sophie Qiu, and David I. August. Hardware multithreaded transactions. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 15–29, New York, NY, USA, 2018. ACM.

[23] John L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[24] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. Helix: automatic parallelization of irregular programs for chip multiprocessing. In *CGO*, 2012.

[27] Jialu Huang, Arun Raman, Yun Zhang, Thomas B. Jablin, Tzu-Han Hung, and David I. August. Decoupled Software Pipelining Creates Parallelization Opportunities. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, April 2010.

[28] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[29] Taewook Oh, Stephen R. Beard, Nick P. Johnson, Sergiy Popovych, and David I. August. A generalized framework for automatic scripting language parallelization. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '17, 2017.

[30] Hanjun Kim. *ASAP: Automatic Speculative Acyclic Parallelization for Clusters*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, September 2013.

[31] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative doall for clusters. *International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[32] Standard Performance Evaluation Corporation (SPEC). `http://www.spec.org/`.

[33] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 179–188, New York, NY, USA, 2005. ACM.

[34] T.N. Vijaykumar, S. Gopal, James E. Smith, and Gurindar Sohi. Speculative versioning cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12):1305–1317, 2001.

[35] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture Code Optimization*, 2(3):247–279, 2005.

[36] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, June 1995.

[37] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[38] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[39] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

[40] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. Design and implementation of transactional constructs for C/C++. In *OOPSLA*, pages 195–212, 2008.

[41] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 20–29, 2002.

[42] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[43] Chen Tian, Min Feng, and Rajiv Gupta. Supporting Speculative Parallelization in the Presence of Dynamic Data Structures. In *Proc. of PLDI*, 2010.

[44] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. *Programming Language Design and Implementation (PLDI)*, June 2012.

[45] ned Productions - nedmalloc. `https://www.nedprod.com/programs/portable/nedmalloc/`.

[46] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 185–192, May 2010.

[47] Milos Prvulovic, María Jesús Garzarán, Lawrence Rauchwerger, and Josep Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 204–215, New York, NY, USA, 2001. ACM Press.

[48] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[49] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sar-

dashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[50] The iPhone XS & XS Max Review: Unveiling the Silicon Secrets. `https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets/2`.

[51] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[52] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 694–701, Piscataway, NJ, USA, 2011. IEEE Press.

[53] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, pages 195–, Washington, DC, USA, 1998. IEEE Computer Society.

[54] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti R. Sarangi, James Tuck, and Josep Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26:80–91, 2006.

[55] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. A unified approach to speculative parallelization of loops in dsm multiprocessors. Technical report, 1998.

[56] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 228–241, New York, NY, USA, 2015. ACM.

[57] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. Emer, and D. Sanchez. Harmonizing speculative and non-speculative execution in architectures for ordered parallelism. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 217–230, Oct 2018.

[58] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, New York, NY, USA, 2006. ACM Press.

[59] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[60] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, June 2005.

[61] M. A. Gonzalez-Mesa, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Effective transactional memory execution management for improved concurrency. *ACM Trans. Archit. Code Optim.*, 11(3):24:1–24:27, August 2014.

[62] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[63] James Reinders. Transactional Synchronization with Intel Core 4th Generation Processor. https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

[64] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: a system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 133–144, New York, NY, USA, 2012. ACM.

[65] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb. 2006.

[66] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, 2007.