Hardware Multithreaded Transactions

Jordan Fix Nayana P. Nagendra Sotiris Apostolakis
Hansen Zhang Sophie Qiu David I. August
Princeton University
{jfix,nagendra,sa8,hansenz,hqiu,august}@princeton.edu

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 equivalently achieve. Instead, this paper 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 99% over sequential execution on a multicore machine with 4 cores.

CCS Concepts • **Computer systems organization** → **Multicore architectures;** • **Software and its engineering** → **Multithreading**

Keywords Hardware transactional memory; multithreaded transactions; pipelined parallelism; thread-level speculation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173172

ACM Reference Format:
Jordan Fix, Nayana P. Nagendra, Sotiris Apostolakis, Hansen Zhang, Sophie Qiu, and David I. August. 2018. Hardware Multithreaded Transactions. In *ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173172

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.

Static parallelization that conservatively respects all potential dependences has achieved success in some domains, such as streaming applications [38], MapReduce applications [8], and embarrassingly parallel DOALL [19] tasks such as matrix multiplication. However, equivalent success has not been found for general purpose, complex programs, such as those with irregular pointer-chasing data structures. To allow for more aggressive TLP extraction without explicit dependence synchronization or communication on these programs, speculative execution and transactional memory (TM) systems have been explored. For example, thread-level speculation (TLS) techniques [5, 7, 13, 34, 35] have not reached widespread use due to their inapplicability or lack of performance.

Alternatively, parallel pipeline techniques [26, 30, 37] and their speculative counterparts [22, 29, 39, 40] can be utilized. These past works have found that speculatively pipeline parallelizing a program often has better performance than other parallelization schemes using traditional TLS.

Unfortunately, most TM systems used for TLS 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 [22, 29] have MTX support, allowing speculative pipeline parallelism techniques to be used on commodity hardware. However, these systems

RIGHTSLINK

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 [4].

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. And without further advances in compiler analyses to eliminate the need for significant amounts of validation checks, these STM systems are not useful for automatic speculative parallelization of complex programs. Prior work [18] 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.

Thus, a TM system with low-overhead speculation validation is essential to achieve automatic parallelization of complex programs and thus 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.

This paper presents the first complete design, implementation, and evaluation of a TM system with support for hardware multi-threaded transactions (HMTXs). 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, 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. To this end, in this system:

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

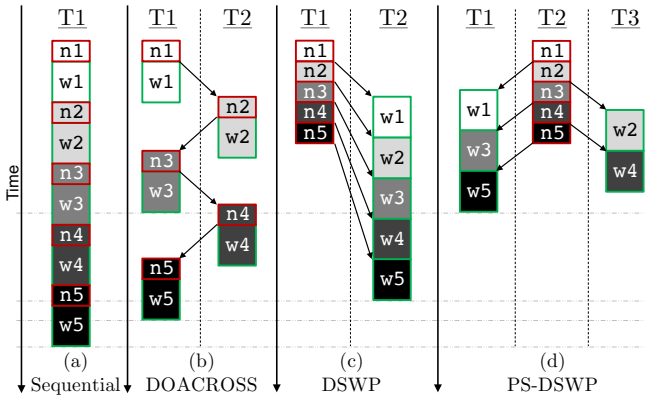

Figure 1. Execution timing diagram of the first 5 iterations of a loop for Sequential, DOACROSS, DSWP, and PS-DSWP.

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 paper presents 8 benchmarks (7 from the SPEC benchmark suite [14, 15], and 1 from MiBench [11]) 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 99% 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.

2 Background and Motivation

2.1 Thread Level Parallelization Techniques

Techniques such as DOALL, DOACROSS [3, 19], and pipeline parallelization [26, 30, 37] 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 at 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.

DOACROSS parallelizes this program in the following fashion. Thread t_1 finds the first node n_1, and then sends the next node n_2 to thread t_2. t_1 continues processing n_1, calling the work function; meanwhile, t_2 can start processing n_2 in

RIGHTSLINK

parallel, and repeat this process by passing n_3 to another thread. This execution model can be seen in Figure 1(b).

This loop could also be pipeline parallelized, for example by using Decoupled Software Pipelining (DSWP)[26], seen in Figure 1(c). Using DSWP, the work of each iteration is separated into a pipeline across multiple threads. In a DSWP parallelized version of the previous linked list example, thread t_1 would iterate on finding every location n_i in the linked list, while sending these locations to t_2 for it to separately process the node by calling the work function.

Parallel-Stage DSWP (PS-DSWP) [16, 30] 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 1(d).

DOACROSS performance depends upon the inter-core latency of the system, because the loop carried dependence must be communicated between threads for every iteration. Meanwhile, pipeline parallelization techniques like 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 [22, 29, 39]. Figure 1 shows that DOACROSS and DSWP could only profitably make use of two threads; meanwhile, PS-DSWP can use many more threads.

2.2 Speculation

Due to the limits of static analysis, it is often hard or impossible for compilers and programmers to find profitable parallelization opportunities. 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 conservatively 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) [5, 7, 12, 34, 35]. 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 [22, 29, 39, 40]. Speculative-DSWP requires multi-threaded transactions (MTXs), wherein transactions can span multiple threads. In DSWP each iteration (wrapped in a transaction) is split across multiple pipelined threads[1], as seen in Figure 1 (c) and (d). Therefore, when a transaction in the first stage of the pipeline makes some speculative modifications to memory, those modifications should be visible to 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 by these pipeline stages from a single transaction should atomically be committed at once.

2.3 Past Multithreaded Transaction Proposals

Vachharajani [39] 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 § 3) which are requirements of an MTX system. While Vachharajani described an initial design for a hardware TM system with MTXs, we are unaware of any detailed implementation or evaluation having been published. Additionally, parts of the design were unrealistic or left incomplete (§ 7).

Later MTX proposals opted for software based TM systems to allow for speculative DSWP execution on commodity hardware [20, 22, 25, 29]. These systems follow in the same path as Vachharajani, assigning a VID to each transaction. They provide for multiple versions of memory by forking the main process (called the *commit process*), which contains and manages the committed non-speculative state, and allowing the other processes to execute transactions, which modify their own versions by relying on copy-on-write support in the OS. 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). 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 [22, 29], 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.

[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. putting n1 and w1 in Figure 1(d) in separate TLS transactions).

RIGHTSLINK

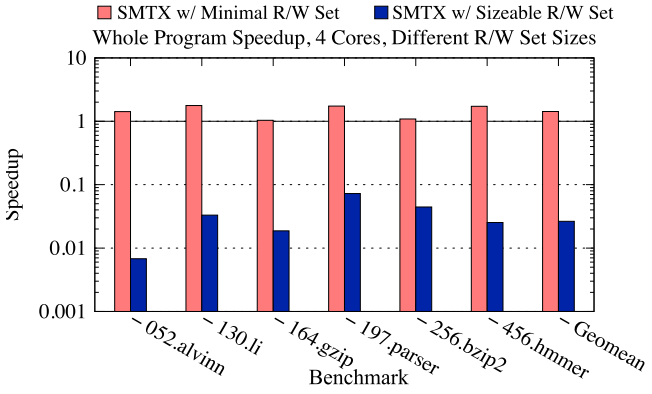

Figure 2. SMTX whole program speedup over sequential execution with a minimal R/W set vs. a substantial R/W set.

Figure 2 demonstrates how heavily performance depends upon read and write sets sizes when using these systems. Whole program speedup is compared for two versions of programs using an MTX STM system: one with a minimal read and write set, and one with speculation validation added to shared data accesses. As expected, more speculation validation turns slight speedups into substantial slowdowns.

Similar results have been presented in the context of automatic speculative parallelization [18, 25]. Johnson [18] found that for simple benchmarks executing in a DOALL fashion using an STM system [20, 21], "imprecise analysis forces the compiler to compensate with more speculation . . . Increased validation overheads cause application slowdown."

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 paper embraces an alternative approach: make speculation validation cheap in order to make speculative parallelization less laborious and more feasible.

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.

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 correspond to the original 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. 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 [9, 12, 32–34, 41], however for MTXs it is a requirement.

To provide support for group transaction commit, all VIDs that are 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 (§ 4.3).

3.1 New HMTX Instructions

New instructions must be added to the instruction set architecture (ISA) in order to support MTX. First, transactions must signal when they begin and end. End does not mean commit; 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: `beginMTX(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 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 `beginMTX` in program order.

Next, a `commitMTX(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

RIGHTSLINK

(a) Original Non-Speculative Sequential Program

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

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

```
vid = 1
leaveLoop = (node == NULL);
while (!leaveLoop):
  beginMTX(vid);
  //create new version of producedNode w/ VID=vid
  producedNode = node; //producedNode is global
  node = node->next;
  leaveLoop = (node == NULL);
  beginMTX(0); //does not commit
  produceVID(vid++);
produceVID(0);
```

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

```
while (vid = consumeVID()):
  beginMTX(vid); //continue TX started in Stage 1
  //finds correct producedNode version w/ VID=vid
  node = producedNode;
  w = work(node); //may modify order of list
  commitMTX(vid);
  if (w > MAX): abortMTX(vid+1);
```

Figure 3. Pseudocode of a speculative version of Figure 1, with sequential (a) and speculative DSWP with MTX (b, c).

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.

An `abortMTX(VID)` instruction is also added in order to explicitly signal an abort 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 `initMTX(pc)` instruction must be used to set the location of this recovery code prior to speculative execution beginning on every thread.

3.2 MTX Instruction Usage

Figure 3 shows a code example comparing the sequential and speculative DSWP versions of a simple linked list traversal program. The DSWP version uses the instructions from § 3.1.

First, note that in the sequential version (Figure 3(a)), the early exit check `if (w > MAX)` limits parallelization; the next parallel iteration cannot begin until the majority of the current iteration's work is complete. Instead, DSWP control flow speculates this dependence does not exist; it is not checked until stage 2, after later iterations (in original program order) have already begun in parallel. If this speculation is incorrect then later transactions are aborted.

Transactions are first started by the initial pipeline stage (Figure 3(b)) via `beginMTX(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 `beginMTX(0)`, which represents that the program is moving back to non-speculative execution, but is **not** committing. All work done between `beginMTX(0)` and `beginMTX(vid)` at the top of the loop is essentially bookkeeping, e.g. sending to stage 2 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, HMTX's versioned memory can be leveraged to communicated each `node` to stage 2 via a single speculative store to the shared location `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 transactions if they are using the same VID (§ 4.1).

Figure 3(c) shows stage 2, where transactions can continue and eventually commit. Via a consume, it determines what VID to use in order to continue with execution of a previously started transaction, and then enters that transaction via `beginMTX(VID)`, just as stage 1. 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 (§ 4.3).

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 `commitMTX(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 (§ 4.3).

3.3 Supporting 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. § 5 discusses solutions to the following problems that would otherwise inhibit performance or make parallelization impossible.

§ 5.1: In processors with large pipelines and branch prediction, loads are often speculatively executed based on branch

RIGHTSLINK

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 lines that are HMTX-based speculatively read. This can result in spurious misspeculations if not resolved, as lines are 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.

§ 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.

§ 5.3: A naïve system would need to keep track of all speculatively accessed lines in order to explicitly transition them on commit or abort. This would require some structure in hardware or software to scale along with the number of accessed lines (which is quite large in the evaluated benchmarks (§ 6.3)), increasing complexity and degrading performance of the system in execution time and energy.

§ 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.

4 Detailed Design of the HMTX System

4.1 Cache Coherence Protocol Modifications

The base design of the system uses a snoopy MOESI cache coherence protocol [36]. 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 have VIDs equal to zero, which is considered to be non-speculative. However when a cache line and/or request have VIDs that are non-zero, hits and misses behave differently. Similar to others [34], these differences are partially dependent upon new "speculative" coherent states: Speculative-Modified (`S-M`), Speculative-Owned (`S-O`), Speculative-Exclusive (`S-E`), and Speculative-Shared (`S-S`). A state diagram for these states is seen in Figure 4 and is explained further in § 4.2.

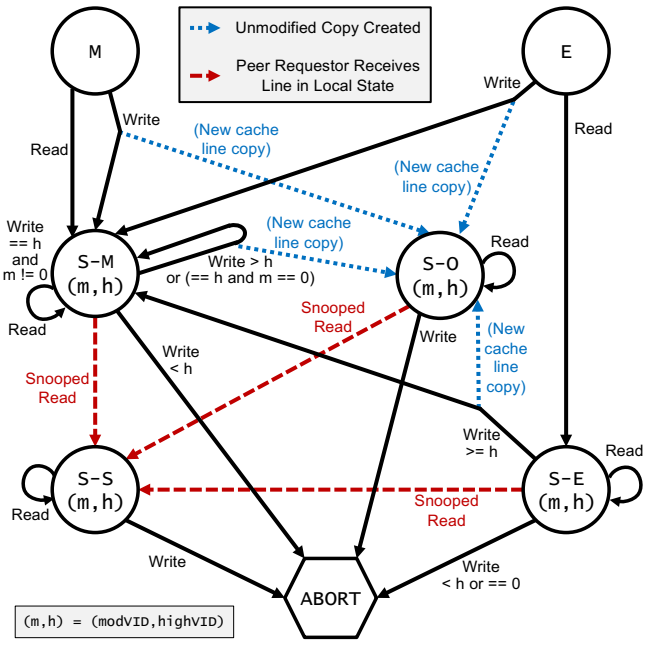

Figure 4. State diagram for speculative accesses. O, S, and I states are not shown for simplicity; they would follow the same path as M or E once acquiring exclusive access.

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). In addition to tag comparison, hits and misses are dependent upon comparing the VID of the request to modVID and highVID. 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. However, note that selecting some speculative versions of cache lines as a victim for writeback past the last level cache forces an abort (§ 5.4).

The modVID corresponds to the VID of the transaction that created this version of the line due to a speculative modification. All non-speculative versions of a line have a modVID of zero. 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. This can be seen in Figure 4, 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.

RIGHTSLINK

The highVID corresponds to the highest VID which accessed this version of the line. This allows reads from later VIDs to access a modified cache line without having to create new cache lines; instead they only need to update highVID. Thus, the use of these two VIDs allows us to represent multiple conceptual cache line versions with a single physical line. 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.

The notation `S-M(m,h)` means that an S-M line has modVID `m` and highVID `h`. The VIDs are always listed in this order when seen in this tuple notation.

Qualitatively, the states can be thought of as follows:

S-M lines represent the "latest" speculative version of the line, meaning this version of the line is the latest with respect to original program order. Thus, if the VID of a speculative write is greater than or equal to highVID, it can proceed without triggering misspeculation, as no "later" access has already occurred to the line. No version of the line exists with a higher modVID. The highest VID to access this line is set to highVID. This version of the line is dirty with respect to main memory, so on commit it must move to a dirty non-speculative state.

S-O lines represent speculatively accessed lines that were later speculatively modified by a write with a higher VID. When such a speculative modification occurs, an unmodified copy is kept in S-O with highVID equal to the VID of the speculative modification, while the new line with speculative modifications moves to S-M. That S-M line may similarly transition to S-O if it is speculatively modified by a write with higher VID. Speculative writes that hit this version of the line trigger an abort due to a potential dependence violation, as some "later" access already occurred to the line.

S-E lines are essentially the same as S-M, except no versions of the line have been modified since entering the cache, whether speculatively or non-speculatively. Consequently, on commit the line returns to a clean non-speculative state (Exclusive or Shared) instead of a dirty non-speculative state (Modified or Owned), preventing unnecessary writeback to memory. This state can never have modVID > 0.

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.

Hits and misses are determined by combining the address and coherent state of the line 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:

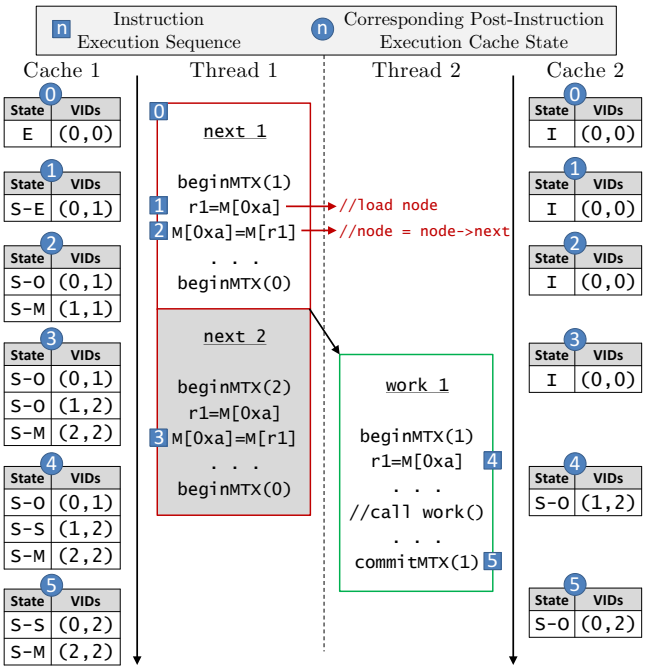

Figure 5. Pseudocode and cache states of the Figure 3 example. Note that cache state is only shown for address `0xa`.

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

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 MOESI.

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 a request incoming 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 knowledge of all versions of this line must be gathered before determining if a line in the local cache should have hit or is a miss. 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 (§ 4.4).

4.2 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. Once the cache has a writable copy, the request can proceed. In the case of a read, the line is moved to S-E

RIGHTSLINK

(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 5 shows an example of this case at instruction 1 with VID 1, resulting in state `S-E(0,1)`.

Now 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 5, with VID 1 at instruction 2, and corresponding new versions `S-O(0,1)` and `S-M(1,1)`.

Continuing with the example in Figure 5, a speculative write executed for VID 2 at instruction 3. In this case the `S-M(1,1)` version transitions to `S-O(1,2)`, setting highVID to 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 4, it is broadcast on the bus and hits the `S-O(1,2)` version of the line due to the scheme as described in § 4.1. The response is sent in `S-O(1,2)`, as seen in Figure 4. If an access with VID greater than or equal to 2 was received it would hit the `S-M(2,2)` version. 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 5 shows the final state after Thread 2 commits. The commit process is explained further in § 4.4.

4.3 Preserving Original Program Semantics

An informal argument is provided that the original program's semantics is preserved using HMTX. VIDs in this system are assigned in order corresponding to the original sequential execution order of the program, thus they can be used to respect dependences during speculative execution, ensuring the original program's semantics is preserved.

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. If this logical order is not respected during execution then then an abort should be triggered.

We will 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. We 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.

Flow Dependences: Assume a store with VID x, s_x, to the same address as a load with VID y, l_y. 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`)`. This is thanks to uncommitted value forwarding. Instead, if l_y occurs first then a version of the line will exist with highVID $== y$ in either `S-M`, `S-E`, or `S-O`. When s_x occurs misspeculation will be detected because of a store to a line with VID $<$ the highVID of the line, y.

Anti-Dependences: Assume a load with VID x, l_x, to the same address as a store with VID y, s_y. 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`. When s_y occurs, it will hit this version of the line because $y >$ the modVID of the line, as $y > x$ and x must be $\geq$ modVID of the line. 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. Instead, if s_y occurs first, then a version of the line will be created in `S-M(`y`,`y`)`, and the original line that it hit will have highVID $== y$. When l_x occurs it will hit that version because $x < y$, and false misspeculation will be avoided.

Output Dependences: Assume two stores with VIDs x, s_x, and y, s_y. 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`)`. Instead, if s_y occurs first, then a version of the line will be created in `S-M(`y`,`y`)`. When s_x occurs misspeculation will be conservatively triggered because of a store to a line with VID $<$ the highVID of the line, y.

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

4.4 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 § 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 (§ 4.7); otherwise behavior of the system is undefined.

A state diagram for commit transitions can be seen in Figure 6. Assume a commit occurs for some VID x. 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 $= 0$. 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`). As seen in Figure 5, a commit occurs at instruction 5, and cache lines transition accordingly.

On an abort for any VID, all uncommitted transactional memory in the cache system is flushed. This facilitates a

RIGHTSLINK

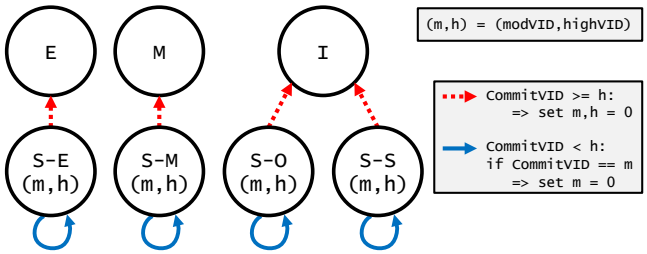

Figure 6. Commit state diagram.

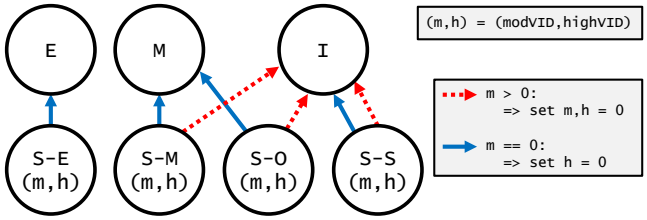

Figure 7. Abort state diagram. Note that `S-E` lines must have modVID $== 0$, hence `S-E` has no transition for modVID > 0.

simpler implementation; aborts should be very rare when a program is parallelized efficiently and thus we optimize for the common case and push slowdowns to the rare abort case.

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 7).

4.5 Efficient VID Comparisons

The majority of the area and power increases (§ 6.4) 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, we 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 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.

4.6 VID Overflow and Reset

Because VIDs are limited to a finite m bits, an issue arises when the system uses all 2^m VIDs. If this occurs, first the software delays all new transactions until the one with VID $= 2^m$ has committed. Next, a VID Reset signal is sent to the memory system, which triggers two actions. First, all caches set LC VID $= 0$. Second, the VIDs on all cache lines are reset to `(0,0)` if an abort was previously seen (§ 5.3). Once this is complete, new transactions can begin with VID $== 1$.

This works thanks to the design of the protocol (§ 4.1); the "latest" versions of the line in `S-M/S-E` will now have VIDs `(0,0)`. This essentially commits them because they have their modVID set to non-speculative state (i.e. $== 0$), and their hit condition simply checks for the access VID $\geq$ modVID. Additionally, the previously buffered "non-latest" speculative versions now in `S-O/S-S(0,0)` can never hit for an access because their hit condition checks for the access VID $<$ highVID which can never be true. These lines will simply be invalidated when selected as a victim.

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. We settled on 6 as a fair medium.

4.7 Operating System and Program Support

Operating system (OS) support is required to determine the maximum size of VIDs, m. The software should query the OS to determine m, and then ensure all outstanding commits have occurred before triggering a VID reset (§ 4.6).

Additionally, the program must ensure that the active VIDs in the system correspond back to original program order, and that commits occur in consecutive order, e.g. VID 2 commits only after 1 and before 3.

Lastly, output must be handled specially inside a transaction. Outputs are explicitly buffered to ensure no speculative effects occur until commit. Prior work [20] created a transactional I/O system to overcome this, which could be used instead.

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 the most important enabling optimizations.

5.1 Squashed Loads and False Misspeculation

As discussed in § 3.3, cache lines may become incorrectly marked as speculatively accessed due to branch misprediction inside of a transaction, leading to spurious misspeculations. To overcome this problem, we introduce the *speculative load acknowledgment* (SLA). 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

RIGHTSLINK

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. This optimization avoids many false misspeculations (§ 6.3).

Note that 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. When a speculative load executes, it is returned to the CPU 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 (§ 6.3).

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.

To support this, we statically link the parallelized programs, and then have the program inform 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.

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.

5.3 Lazy Commits and Aborts

As noted in § 3.3, a naïve scheme such as that presented in § 4.4 would need to keep track of all speculatively accessed lines in a manner which is not scalable or efficient for large read and write sets. To improve upon this scheme, the HMTX system adapts an approach used by other works [13, 27, 41] by using *lazy speculative state processing*. A new register is added to each cache representing the latest committed VID (LC VID). Additionally, two new bits are added to every cache line: a Committed Bit (CB) and an Aborted Bit (AB).

On commit, all caches must complete two tasks. First, they set their LC VIDs to the VID of the commit. Second, every line has its CB set to 1 if AB $\neq 1$. Meanwhile, aborts do not change LC VID, but similar to commits each line sets its AB to 1 if CB $\neq 1$. This ensures that AB and CB are never simultaneously set.

For every speculative access that arrives, the cache continues using the same hit and miss logic as before, as described in § 4.1. Non-speculative accesses use VID = LC VID for hit logic only. This intuitively makes sense, as non-speculative accesses should access the last committed version of a line.

Using CB and AB, lines are updated for commit (if CB $== 1$) or abort (if AB $== 1$) if a hit occurs for that version of the line, or if that line is chosen as a victim for write back to the next level. For commits, the VIDs of the line will be checked against the cache's LC VID, and any transitions necessary can be made on a single line basis. Similarly, aborts can transition based on the line's VID. Lastly, processing aborts lazily requires that we additionally ensure that hits can never occur for lines with modVID > 0 and AB $== 1$.

Given this design, no complex cache operations are required on commit or abort; execution can proceed quickly without waiting for an expensive or costly cache operation where every line must be explicitly transition based on the commit or abort state machine. This additionally limits the complexity of implementation. This lazy commit scheme is possible because of the design of the coherence states, which allows for a lines to transition to its next state without needing to query for the state of any other lines in the system.

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, we must guarantee 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. 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.

If a line not in state `S-O` with modVID $= 0$ is selected as a victim from the last level cache, then the transaction should abort. In order to reduce such aborts, victim selection in the last level cache can prioritize such `S-O` lines for eviction over other speculatively accessed lines.

RIGHTSLINK

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%	2,290,717	0.158	1.28%	11.5%	0.245%
130.li	PS-DSWP	100%	181,844,120	22.5	4.21%	20.5%	3.65%
164.gzip	PS-DSWP	98.4%	6,248,356	3.32	7.08%	14.6%	2.68%
186.crafty	PS-DSWP	99.5%	4,498,903	1.50	4.92%	13.1%	5.59%
197.parser	PS-DSWP	100%	24,733,144	24.6	2.56%	19.2%	1.05%
256.bzip2	PS-DSWP	98.5%	131,271,380	17.3	6.04%	12.6%	1.33%
456.hmmer	PS-DSWP	100%	1,709,195	0.187	1.40%	4.83%	1.03%
ispell	PS-DSWP	86.5%	43,752	0.0280	13.0%	16.6%	2.82%

Table 1. Statistics from simulated speculative execution using HMTX, and from native sequential non-speculative execution.

Feature	Parameter
Architecture	Alpha 21264
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 2. Architectural Configuration in gem5.

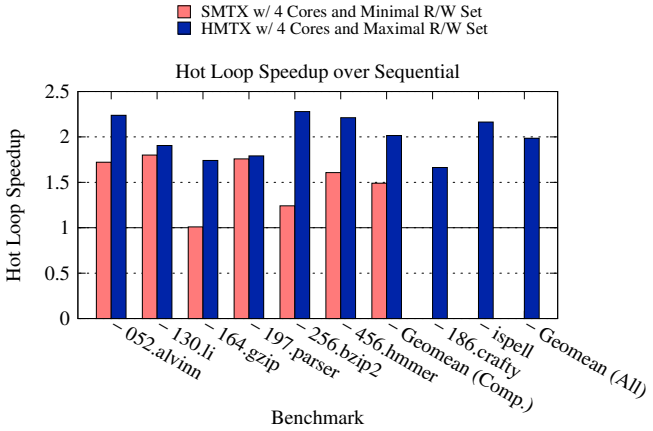

Figure 8. Hot loop speedup over sequential using 4 cores. SMTX versions have minimal read and write sets due to expert manual transformation. HMTX versions perform speculation validation on *every read and write* inside a transaction, i.e. the maximum possible read and write set. Note that we have no SMTX comparison for 186.crafty and ispell; accordingly, "Comp." represents those benchmarks with an SMTX comparison, while "All" represents all benchmarks.

6 Evaluation

The HMTX system was implemented and evaluated using the gem5 simulator [2] in full system mode with a 4-core out-of-order processor. Table 2 shows the hardware configuration. Hot loop speedup is compared. Approximately 15% of the iterations of the hot loops are measured and evaluated due to limitations of the simulation environment.

6.1 Benchmarks

We evaluated 8 benchmarks (7 from the SPEC benchmark suite, and 1 from MiBench), all of which need speculation for efficient TLP transformation. Of these benchmarks, 6 were also evaluated by SMTX [29]; replicated SMTX results for these 6 are directly compared against. We focused mostly 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 1 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.

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 (§ 2.2).

6.2 Hot Loop Speedup

As seen in Figure 8, the HMTX system with 4 cores provides a geomean speedup of 1.99x over sequential execution on all 8 benchmarks. On the 6 benchmarks evaluated by both HMTX and SMTX, HMTX has a speedup of 2.02x, outperforming SMTX with a speedup of 1.44x. Thus, with 4 cores HMTX achieves better performance than SMTX while also performing significantly more speculation validation. Note that SMTX requires the extra commit process, taking up one core's resources.

RIGHTSLINK

Hardware	Exec Model	Area (mm^2)	Total Leakage (W)	Geomean Runtime Dynamic (W)	Geomean Energy (J)
Commodity	Sequential (All)	107.1	5.515	3.577	7.323
	Sequential (Comp.)			3.654	10.91
	SMTX, Min R/W			13.66	15.32
Commodity + HMTX Extensions	Sequential (All)	111.1	5.607	3.618	7.431
	Sequential (Comp.)			3.696	11.07
	SMTX, Min R/W			13.87	15.57
	HMTX, Max R/W (All)			14.43	8.088
	HMTX, Max R/W (Comp.)			14.46	11.77

Table 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 energy between "Comp." and "All" is largely due to the short execution time of ispell compared to other benchmarks.

As shown in Figure 2, when the SMTX versions perform more speculation validation (though still less than the HMTX versions), its performance degrades badly. Therefore, these results show that the HMTX system is a large step toward enabling automatic parallelization.

6.3 Misspeculation

No misspeculation occurred in any of the benchmarks that were evaluated, as only high confidence speculation is performed. All false misspeculation was avoided, such as those prevented thanks to SLAs (§ 5.1). The number of avoided misspeculations, seen in Table 1, varies for each benchmark depending upon the data access patterns given their complex data structures and control flow. 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.

Table 1 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.

An abort could also be triggered if certain speculatively modified lines overflow the caches (§ 5.4). 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.

Figure 9 shows the average size of the read and write sets. The geomean combined set size is 957 kB. The benchmark with the largest average size was for 256.bzip2, with 16,222 kB. Additionally, the large number of speculative memory accesses per transaction can be seen in Table 1.

6.4 Area, Power, and Energy

Area and power are modeled with McPAT [23]. The 22nm technology node is used. Power gating and low L2 cache standby power are utilized. Table 3 displays statistics gathered.

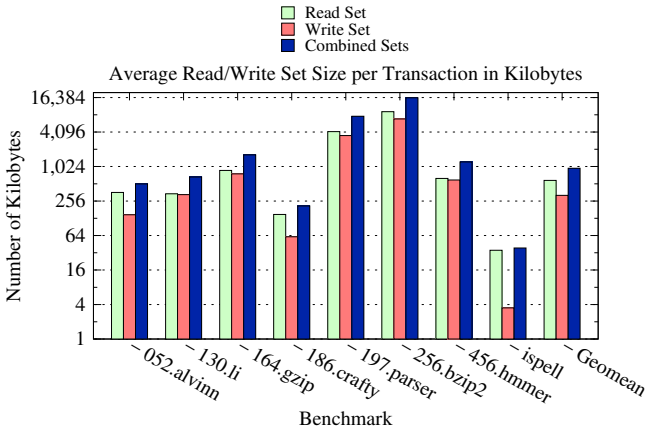

Figure 9. Average size of the read and write sets in kB.

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 § 4.5.

McPAT uses CACTI [24] in order to model caches, which performs architectural modeling of SRAM based caches. Total leakage increases marginally when adding in HMTX extensions (Table 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, we ran the same SMTX and sequential benchmarks 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 3). This highlights the low impact of HMTX extensions.

RIGHTSLINK

7 Related Work

7.1 MTX by Vachharajani [39]

The HMTX system follows Vachharajani's lead by adding version IDs to each cache line, as noted in §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 §4.4). Even with an ORB-like structure [34] 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.

By 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 or abort (§ 4.4). This allows for both to occur lazily, similar to other works [27, 41]. 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.

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. By 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. By contrast, HMTX uses cache line-level granularity. This reduces the complexity of implementation without any increase in misspeculation rates on the evaluated benchmarks.

7.2 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 [7, 10, 27, 31, 32, 34, 41]. 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 (§ 2.3).

Many past systems provide an ordering for transactions as HMTX does, allowing for uncommitted value forwarding as an optimization [9, 12, 17, 32–34, 42]. However, as noted in § 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 [1, 6, 27, 28], 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.

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

8 Conclusion and Future Work

This paper presented the HMTX system, the first complete design, implementation, and evaluation of a hardware TM system with support for multi-threaded transactions (MTX). MTX is required for thread-level pipeline parallelization, an important class of parallel execution techniques. 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 99% is achieved over sequential execution, with modest increases in power and energy consumption.

Future work could adapt the HMTX coherence scheme to a directory-based protocol to allow for efficient scaling to many more cores. Additionally, similar to prior systems [27], unlimited read and write sets could be supported by overflowing speculatively modified versions of lines into memory and managing them via data structures.

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

Acknowledgments

We thank the Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments and suggestions. This work is supported by the National Science Foundation (NSF) under Grants OCI-1047879, CCF-1439085, and CNS-0964328. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the Liberty Research Group and do not necessarily reflect the views of the NSF. This work was carried out when the authors were working at Princeton University.

RIGHTSLINK

References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, USA, 316–327. https://doi.org/10.1109/HPCA.2005.41

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

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

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6, 5 (2008), 46–58. https://doi.org/10.1145/1454456.1454466

[5] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA.2006.13

[6] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. 2006. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, USA, 347–358. https://doi.org/10.1145/1168857.1168901

[7] Marcelo Cintra, José F. Martínez, and Josep Torrellas. 2000. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 13–24. https://doi.org/10.1145/339647.363382

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

[9] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. 2005. Tradeoffs in Buffering Speculative Memory State for Thread-level Speculation in Multiprocessors. *ACM Transactions on Architecture Code Optimization* 2, 3 (2005), 247–279. https://doi.org/10.1145/1089008.1089010

[10] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. 1998. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA '98)*. IEEE Computer Society, Washington, DC, USA, 195–. http://dl.acm.org/citation.cfm?id=822079.822729

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

[12] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, USA, 58–69. https://doi.org/10.1145/291069.291020

[13] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. http://dl.acm.org/citation.cfm?id=998680.1006711

[14] John L. Henning. 2000. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer* 33, 7 (July 2000), 28–35. https://doi.org/10.1109/2.869367

[15] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

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

[17] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/2830772.2830777

[18] Nick P. Johnson. 2015. *Static Dependence Analysis in an Infrastructure for Automatic Parallelization*. Ph.D. Dissertation. Department of Computer Science, Princeton University, Princeton, New Jersey, United States.

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

[20] Hanjun Kim. 2013. *ASAP: Automatic Speculative Acyclic Parallelization for Clusters*. Ph.D. Dissertation. Department of Computer Science, Princeton University, Princeton, New Jersey, United States.

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

[22] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. 2010. Scalable Speculative Parallelization on Commodity Clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[23] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. 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)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[24] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. 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)*. IEEE Press, Piscataway, NJ, USA, 694–701. http://dl.acm.org/citation.cfm?id=2132325.2132479

[25] Taewook Oh, Stephen R. Beard, Nick P. Johnson, Sergiy Popovych, and David I. August. 2017. A Generalized Framework for Automatic Scripting Language Parallelization. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '17)*.

[26] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 105–118. https://doi.org/10.1109/MICRO.2005.13

[27] Milos Prvulovic, María Jesús Garzarán, Lawrence Rauchwerger, and Josep Torrellas. 2001. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM Press, New York, NY, USA, 204–215. https://doi.org/10.1145/379240.379264

[28] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*. 494–505.

RIGHTSLINK

[29] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative Parallelization Using Software Multi-threaded Transactions. In *Proceedings of the Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[30] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David I. August. 2008. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*.

[31] Jose Renau, Karin Strauss, Luis Ceze, Wei Liu, Smruti R. Sarangi, James Tuck, and Josep Torrellas. 2006. Energy-Efficient Thread-Level Speculation. *IEEE Micro* 26 (2006), 80–91. Issue 1.

[32] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. 2005. Tasking with Out-of-order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05)*. ACM, New York, NY, USA, 179–188. https://doi.org/10.1145/1088149.1088173

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

[34] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. 2000. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*. 1–12.

[35] J. Gregory Steffan and Todd C. Mowry. 1998. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. 2–13.

[36] P. Sweazey and A. J. Smith. 1986. 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)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 414–423. http://dl.acm.org/citation.cfm?id=17407.17404

[37] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. 2007. 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)*. 356–369. https://doi.org/10.1109/MICRO.2007.7

[38] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 12th International Conference on Compiler Construction*.

[39] Neil Vachharajani. 2008. *Intelligent Speculation for Pipelined Multithreading*. Ph.D. Dissertation. Department of Computer Science, Princeton University, Princeton, New Jersey, United States.

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

[41] T.N. Vijaykumar, S. Gopal, James E. Smith, and Gurindar Sohi. 2001. Speculative Versioning Cache. *IEEE Transactions on Parallel and Distributed Systems* 12, 12 (2001), 1305–1317. https://doi.org/10.1109/71.970565

[42] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. 1998. *A Unified Approach to Speculative Parallelization of Loops in DSM Multiprocessors*. Technical Report.

RIGHTSLINK