# BRIDGING THE GAP FOR HARDWARE TRANSACTIONAL MEMORY

A Dissertation
Presented to
The Academic Faculty

By

Sunjae Young Park

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2018

**BRIDGING THE GAP FOR HARDWARE TRANSACTIONAL MEMORY**

Approved by:

Dr. Milos Prvulovic
School of Computer Science
*Georgia Institute of Technology*

Dr. Hyesoon Kim
School of Computer Science
*Georgia Institute of Technology*

Dr. Santosh Pande
School of Computer Science
*Georgia Institute of Technology*

Dr. Moinuddin Qureshi
School of Electrical and Computer Engineering
*Georgia Institute of Technology*

Dr. Christopher J. Hughes
*Intel*

Date Approved: October 16, 2018

## ACKNOWLEDGEMENTS

First, I thank my parents, who patiently waited for me to complete the process and helped me stay motivated. Both my mother and father have trusted me through the years, patiently waiting for me and encouraging me to never give up.

Many thanks to my wife Seonghye, for supporting me through the final years. You have been a jolt of new energy when I felt like I was spinning my wheels without going anywhere.

Thank you Junseok for bringing new joy to my life. It always lifts me up to see you running towards me when I pick you up from day care.

Thanks to my new daughter-to-be, who has already changed my life for the better.

Thanks to my brother Sunjin, for always caring for me and helping me out.

Thanks to my advisor Professor Milos Prvulovic who have been a great inspiration for all of my research, constantly showing me the way forward whenever I felt I had lost my way. There have been numerous cases where I had been able to overcome my researcher's block by listening to your ideas.

I thank my co-author and mentor Dr. Christopher Hughes for consistently being supportive of me during the frequent meetings we had, giving me important feedback and ideas on how to move forward.

And last, thanks to God for guiding me throughout my entire life, and protecting me, keeping me healthy and strong.

## TABLE OF CONTENTS

**Acknowledgments** . . . . . iii

**List of Tables** . . . . . vii

**List of Figures** . . . . . viii

**Chapter 1: Introduction and Background** . . . . . 1

1.1 Limitations of Conventional HTM and Outline of Thesis . . . . . 4

**Chapter 2: PleaseTM: Enabling Transaction Conflict Management in Requester-wins Hardware Transactional Memory** . . . . . 7

2.1 Background . . . . . 7

2.1.1 Conflict Handling under Conventional HTMs . . . . . 7

2.1.2 Prior Solutions . . . . . 10

2.2 PleaseTM . . . . . 11

2.2.1 Overview . . . . . 11

2.2.2 Write-write Conflict with Plea Accepted . . . . . 14

2.2.3 Write-write Conflict with Plea Denied . . . . . 15

2.2.4 Read-write Conflict with Plea Accepted . . . . . 16

2.3 Additional Conflict Resolution Policies . . . . . 17

2.4 Correctness . . . . . 19

2.4.1 ABA Problem . . . . . 20

2.5 Experimental Results . . . . . 21

2.5.1 Setup . . . . . 21

2.5.2 Overall Results . . . . . 22

2.5.3 More Conflict Resolution Policies . . . . . 25

2.5.4 Overhead of Avoiding Coherence Protocol Modifications . . . . . 27

2.5.5 Effect on Software Fallback Thresholds . . . . . 28

2.5.6 Conflict Resolution and Software Fallback . . . . . 30

2.5.7 Conflict Resolution and Software Optimizations . . . . . 32

2.6 Related Work . . . . . 33

**Chapter 3: Transactional Pre-abort Handlers in Hardware Transactional Memory . . . . . 36**

3.1 Background . . . . . 36

3.2 Pre-abort Handlers . . . . . 38

3.2.1 Supported Abort Types . . . . . 40

3.2.2 Hardware Support . . . . . 41

3.3 Pre-abort Handler Use Cases . . . . . 43

3.3.1 Handling System Call Aborts . . . . . 43

3.3.2 Handling Capacity Aborts . . . . . 46

3.3.3 Inserting Non-transactional Work . . . . . 48

3.3.4 Pausing Transactions on Fallback-lock Activity . . . . . 48

3.3.5 Pausing Transactions on Critical Section Conversion . . . . . 51

3.3.6 Other Possibilities . . . . . 52
3.4 Evaluation Setup . . . . . 52
3.4.1 Microbenchmark Results . . . . . 54
3.4.2 Non-transactional Work Results . . . . . 56
3.4.3 STAMP Results . . . . . 57
3.5 Related Work . . . . . 60

**Chapter 4: Forgive-TM: Supporting Lazy Conflict Detection in Eager Hardware Transactional Memory . . . . . 63**

4.1 Background . . . . . 63
4.2 Related Work . . . . . 65
4.3 Forgive-TM . . . . . 67
4.3.1 Hardware Overview . . . . . 69
4.3.2 LazySet Maintenance . . . . . 71
4.3.3 Scoring Mechanism . . . . . 72
4.3.4 Operation Flowcharts . . . . . 73
4.3.5 Examples . . . . . 75
4.3.6 Discussion . . . . . 77
4.4 Evaluation and Results . . . . . 79

**Chapter 5: Conclusion . . . . . 86**

**References . . . . . 95**

## LIST OF TABLES

3.1 Abort Types with Pre-abort Handlers . . . . . 40

4.1 Types of Data Conflicts . . . . . 78

4.2 Benchmark Characteristics * indicates average per transaction . . . . . 79

## LIST OF FIGURES


1.1 Baseline HTM Architecture . . . . . 3

2.1 Timeline of wasted work in requester-wins HTM . . . . . 8

2.2 Architecture for PleaseTM . . . . . 13

2.3 Write-write conflict with T0's plea accepted . . . . . 14

2.4 Write-write conflict with T0's plea denied . . . . . 15

2.5 Read-write conflict with plea accepted . . . . . 17

2.6 A write-write conflict with a different conflict resolution policy . . . . . 18

2.7 Timeline for A → B → A . . . . . 21

2.8 Baseline (*RequesterWins*) vs. Two Simple Plea-enabled Policies . . . . . 23

2.9 Ratio of Friendly-fire Aborts . . . . . 24

2.10 Speedup of Various Types of Plea Bits Compared to *ResponderWins* . . . . 26

2.11 Performance Overhead of Avoiding Coherence Modifications . . . . . 28

2.12 Overhead Messages (Refetch/Nack) . . . . . 29

2.13 Speedup With Different Fallback Thresholds . . . . . 30

2.14 Speedup Chart Various Software Fallbacks . . . . . 31

2.15 Speedup Chart with Optimized Software . . . . . 32

3.1 Life of a Transaction with the Baseline Fallback . . . . . 37

3.2 Flowchart With Pre-abort Handler . . . . . 39

3.3 Pre-abort Handler For System Calls . . . . . 45

3.4 Pre-abort Handler for Fallback-lock Conflicts . . . . . 49

3.5 System Call Microbenchmark Results . . . . . 54

3.6 Capacity-scaled Microbenchmark Results . . . . . 55

3.7 Adding non-transactional logging to vacation . . . . . 57

3.8 Overall Results . . . . . 57

3.9 Contribution of Capacity Aborts . . . . . 58

3.10 Other Statistics . . . . . 59

3.11 Overhead of Pre-abort Handlers . . . . . 60

4.1 Conflict Detection . . . . . 64

4.2 Architecture . . . . . 70

4.3 Operation Flowcharts . . . . . 73

4.4 Timeline of Writes in Forgive-TM . . . . . 75

4.5 Other Scenarios . . . . . 76

4.6 Overall Results . . . . . 80

4.7 Abort Types . . . . . 81

4.8 # Eager Aborts Triggered by Each Address X axis is sorted by #aborts from baseline . . . . . 82

4.9 Sensitivity Analysis With the LazySet . . . . . 83

4.10 Ratio of Useful Work Compared to Baseline . . . . . 84

## SUMMARY

Transactional memory (TM) is a promising new tool for shared memory application development. Unlike mutual exclusion locks, TM allows atomic sections to execute concurrently, potentially resulting in improved performance. Commercial releases of hardware TM (HTM) promises this functionality to end users.

However, the commercial implementations work to provide TM functionality with the minimum amount of hardware changes required, unlike research prototypes that can work from a clean slate. As a result, there are significant gaps in performance of the commercial implementations compared to those proposed by the research community. In this thesis, I propose to several ideas that keep with this mindset, but still close the gap in performance.

First, I introduce plea bits that can be used to provide enhanced conflict resolution policies, compared to the basic "requester-wins" policy used in commercial HTM implementations. These bits are only used by the HTM hardware and ignored by the coherence protocol, reducing verification overhead that can result from modifying the protocol, which past work has opted for.

Second, I propose stopping automatic state rollback when the transaction encounters an abort-causing condition, and calling a pre-abort handler instead. HTMs are forced to abort valid transactions due to various limitations. By this simple change however, programmers can insert various code that mitigates such limitations, or even insert non-transactional work within the transaction.

Last, I propose to change how speculative writes are handled within the transaction, allowing for lazy conflict detection on an eager conflict detection HTM, which is what commercial HTM use. Lazy conflict detection promises better performance, but have been viewed to require changes to the coherence protocol. Instead, by defering the write permission acquisition to the end of the transaction, I show that it is possible to support lazy conflict detection without such extensive changes.

# CHAPTER 1
# INTRODUCTION AND BACKGROUND

Mutex locks have long been used as a mechanism to provide atomicity in shared memory programs. A thread that accesses a shared object in memory is required to first acquire the mutex associated with that object. The lock allows only one thread at a time to hold it, thus ensuring atomicity for an operation on the shared object. To simplify programming, a lock typically protects more than one memory location. This can result in unnecessary serialization on the lock even when different threads perform operations on disjoint sets of memory locations, i.e. when the operations could have been performed concurrently yet atomically. Thus locks represent a pessimistic concurrency control mechanism.

In contrast, transactional memory (TM) provides optimistic concurrency control. TM allows concurrent transactions to speculatively execute code that accesses shared memory. Transactions for which no conflict is detected are allowed to commit their work, avoiding unnecessary serialization. When a conflict is detected, e.g. when two concurrent transactions try to write to the same memory location, the conflict must be resolved, typically by delaying or aborting one of the conflicting transactions.

Until recently, only software-based TM systems have been available to users, whereas hardware transactional memory (HTM) was only a (very active) research topic. This has changed with the introduction of several commercial HTM offerings [1, 2, 3]. However, while much HTM research has focused on sophisticated HTM proposals that provide strong guarantees and extensively change the coherence protocol (or completely replace it with TM-specific coherence [4]), the commercial offerings mostly use much simpler implementations, often dubbed "best-effort" for their lack of any guarantees that an HTM transaction will ever commit[1], and with a "requester-wins" approach to conflict resolution that does

[1] Exceptions to this exist, but place severe restrictions on the operations that can be placed in a transaction.

not require any changes to the underlying coherence protocol [5].

In these HTMs, data conflict detection relies on the coherence protocol. When a transaction needs to access a cache block, the block is requested using the unmodified coherence protocol. If this block is held by another core in a conflicting coherence state, that core gets an invalidation or downgrade request that forces it to change its coherence state.

A transaction running on the core receiving such a request must be aborted. In other words, the "requester" transaction has "won" the conflict. However, such a policy can result in aborting transactions that have done more work instead of those that have done less work, and creates a strong bias against long running transactions that have already accessed many blocks. The long running transactions are "vulnerable" to aborts from many small-/young transactions that may access one of those blocks. Thus requester-wins HTMs can experience performance degradation when there are many data conflicts between transactions.

The reason best-effort, requester-wins HTMs are popular in commercial offerings is that they are easier to implement. However, even supposedly simple HTM designs can be difficult to implement correctly [6]. This argues for focusing performance improvement efforts on this style of HTM, rather than on more complex HTMs that are less likely to be used in practice.

Figure 1.1 shows a high-level depiction of our baseline architecture. The *HTM Engine* contains hardware that manages the transaction, such as the *Abort Pending* bit that signals whether the running transaction has been aborted, and the *Abort Hardware* that rolls back updates by the aborted transaction. In the private cache, each block has an additional *Transactional (tnx)* bit, which indicates whether the line has been accessed by the transaction (thus adding them to the read or write set).

Coherence messages are sent through the *Coherence Layer*, which also contains the *Directory*. To request read or write permissions, cores send standard coherence protocol messages down to the *Coherence Layer*. When a coherence invalidate or downgrade mes-

sage for a cache block with its *tnx* bit set arrives, this means that there has been a data conflict on that address. This sets the *Abort Pending* bit, triggering the transaction abort.

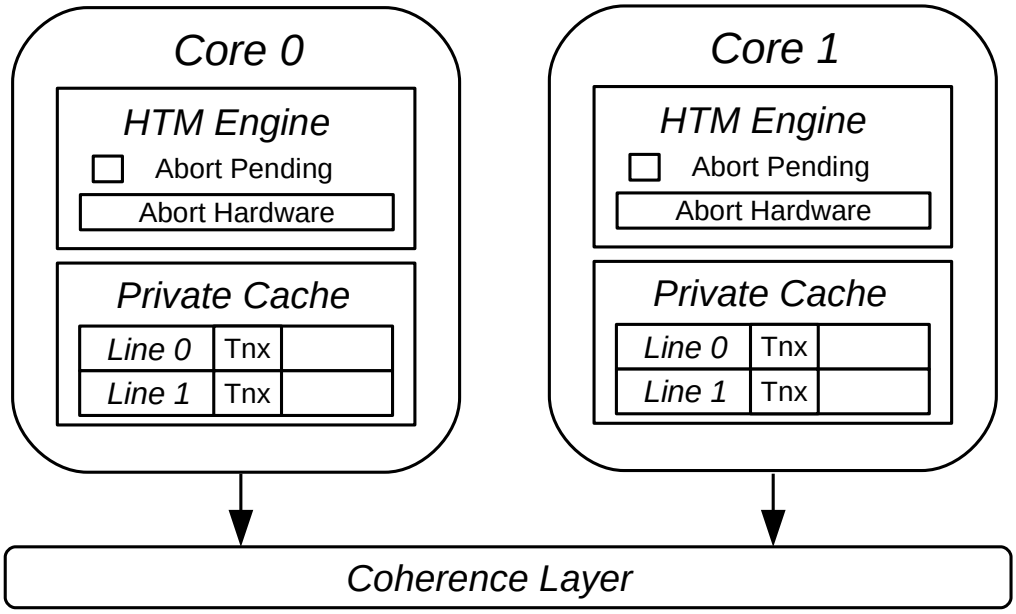



Figure 1.1: Baseline HTM Architecture

This type of best-effort HTM has been the dominant mechanism in providing TM support for commercial offerings [1, 7, 2, 3, 8]. The implementation is relatively low-cost and when there is little data contention, best-effort HTMs perform very well. Data conflict checking is done at access time, which means that there is little work required at commit time.

More importantly, requester-wins HTMs leave the coherence protocol unchanged. There are no additional message types or coherence states that can complicate the coherence protocol and its verification. In other words, the coherence protocol is unaware of the HTM.

However, with data conflicts, best-effort HTMs can have lower performance than more sophisticated HTMs. One problem is that requester-wins HTMs often penalize older transactions that have done more work instead of younger ones. When a transaction conflicts with another, older, transaction by requesting data already accessed by the older one, the core running the younger transaction is the requester, and the core with the older one is

the responder. The responder core has no option except to honor the request, aborting the older transaction. In other words, the older transaction "loses" against the younger one. However, this is often suboptimal because the older transaction has done more work than the younger one, and this makes it harder for large transactions to succeed in the presence of many small transactions.

## 1.1 Limitations of Conventional HTM and Outline of Thesis

conventional hardware TM implementations are designed in this fashion to avoid making any changes to the coherence protocol. When a transaction needs to access a cache block, the block is requested using the unmodified coherence protocol. If this block is held by another core in a conflicting coherence state, that core gets an invalidation or downgrade request that forces it to change its coherence state. However, there can be several shortcomings with this approach.

A transaction running on the core receiving such a request must be aborted. In other words, the "requester" transaction has "won" the conflict. However, such a policy can result in aborting transactions that have done more work instead of those that have done less work, and creates a strong bias against long running transactions that have already accessed many blocks. The long running transactions are "vulnerable" to aborts from many small-/young transactions that may access one of those blocks. Thus requester-wins HTMs can experience performance degradation when there are many data conflicts between transactions.

In addition, because coherence protocol messages are used to detect data conflicts, the conflict detection scheme in conventional HTMs are called "eager." Data conflicts are detected immediately, as soon as the other thread accesses the said data. A different style of data conflict detection is called "lazy conflict detection." Here, transactions do reads and writes locally, without notifying the other threads. Only when the transaction is ready to commit are the read and write sets exposed to the others. Because conflict detection is done

only at the very end, the window of vulnerability is much shorter when using lazy conflict detetion.

Also, conventional hardware TM implementations can also experience other, non-conflict aborts as well. For example, speculative data is commonly buffered in a private buffer (such as the private cache), which isolates it from other cores until the transaction is committed [2, 3, 1]. If the speculative data does not fit within this private buffer, the transaction is unable to continue because the HTM implementation will no longer be able to roll back if a real data conflict occurs, so the HTM transaction aborts a transaction upon detecting a private-buffer overflow. Similarly, a system call result in aborting a transaction for which no actual conflict has been detected, and no conflict may actually happen at all [9, 2, 3, 1].

As noted earlier, these HTM systems are also *best effort*, in that they make no forward progress guarantees. As a result, software needs to provide alternative means of providing forward progress, called the fallback path [2]. One common implementation of the fallback path is to execute the transaction as a critical section, whose mutual exclusion is enforced by aborting all other active transactions, even though no data conflicts have been detected.

The reason best-effort, requester-wins HTMs are popular in commercial offerings is that they are easier to implement. However, even supposedly simple HTM designs can be difficult to implement correctly [6]. This argues for focusing performance improvement efforts on this style of HTM, rather than on more complex HTMs that are less likely to be used in practice. In this thesis, we discuss three such techniques.

In Chapter 2, we show that it is possible to follow the key tenet of requester-wins HTMs (*thou shalt not modify the coherence protocol*) while allowing more sophisticated conflict resolution policies. The solution we propose, which we call PleaseTM, is to insert a plea bit (or bits) in each coherence response. These plea bits are transparent to the coherence protocol (no effect on coherence states, transitions, or verification), and the only change is the addition of these bits into the coherence message. The plea bit allows a responder core running a transaction to inform the requester that it will abort a transaction if the requester

proceeds. Using this information, the requesting transaction can implement the actual conflict resolution policy: it can choose to ignore the request and proceed (the requester won), but it can also choose to abort itself before using or modifying the block (the requester gives up, or loses), allowing the responding transaction to avoid the abort.

Later, in Chapter 3, we propose a generic mechanism that can be used to support mitigating actions, and possibly other purposes. Thus we propose that the HTM, instead of simply aborting the transaction upon detecting an action that creates an an abort-causing condition, instead trigger execution of a *pre-abort software handler* without aborting the transaction [2]. This handler would execute outside of the speculative context, and can implement mitigating actions that allow the transaction to continue, or simply allow the transaction to be aborted, e.g. when abort is actually needed or when the appropriate mitigating action is not implemented in the handler. To provide the handler with the information necessary to make decide which action to take, the type of the abort trigger and additional arguments such as relevant instruction and data addresses is passed to the handler.

Then, in Chapter 4, we go back to the issue of conflict detection (as opposed to conflict resolution, which we discuss in Chapter 2). We propose a technique that allows requester-wins HTMs to *emulate* lazy conflict detection. When a transaction needs to issue a speculative write, it does so with read permission (GetS), instead of write permission (GetX) as normally done. Later, when the transaction is ready to commit, the HTM hardware steps through each speculative but dirty line, and requests write permission for each. Transactional reads on the other hand are done eagerly as usual.

Lastly, we conclude the thesis in Chapter 5.

[2]Note that this is different from the regular abort handler, which is called *after* the transaction is aborted

# CHAPTER 2

# PLEASETM: ENABLING TRANSACTION CONFLICT MANAGEMENT IN REQUESTER-WINS HARDWARE TRANSACTIONAL MEMORY

## 2.1 Background

### 2.1.1 Conflict Handling under Conventional HTMs

As discussed earlier, conventional HTMs detect data conflicts by using coherence protocol messages [10]. When a transaction that speculatively read from an address receives an invalidation request (`GetM` request), this indicates a read/write conflict with another core. Likewise, when a transaction that did a speculative write receives a coherence request, this indicates a data conflict with another core. As a result of the coherence message, the transaction is subsequently aborted.

This type of best-effort HTM has been the dominant mechanism in providing TM support for commercial offerings [1, 7, 2, 3, 8]. The implementation is relatively low-cost and when there is little data contention, best-effort HTMs perform very well. Data conflict checking is done at access time, which means that there is little work required at commit time.

More importantly, requester-wins HTMs leave the coherence protocol unchanged. There are no additional message types or coherence states that can complicate the coherence protocol and its verification. In other words, the coherence protocol is unaware of the HTM.

However, with data conflicts, best-effort HTMs can have lower performance than more sophisticated HTMs. One problem is that requester-wins HTMs often penalize older transactions that have done more work instead of younger ones. When a transaction conflicts with another, older, transaction by requesting data already accessed by the older one, the core running the younger transaction is the requester, and the core with the older one is

the responder. The responder core has no option except to honor the request, aborting the older transaction. In other words, the older transaction "loses" against the younger one. However, this is often suboptimal because the older transaction has done more work than the younger one, and this makes it harder for large transactions to succeed in the presence of many small transactions.

Figure 2.1 shows a simple example that illustrates this. The example uses two cores, `C0` and `C1`. The directory is shown as `DIR`. A transaction execution is indicated by a solid line in a thread's timeline, whereas execution outside a transaction is shown as a dotted line. An empty circle shows when a transaction starts, and a filled circle shows commit. A filled diamond shows when a transaction has aborted. Actions taken by the core or the cache controller are indicated by gray boxes. For reference, the lower part of the figure also shows how the cache's coherence state changes at each core (using the MESI protocol).

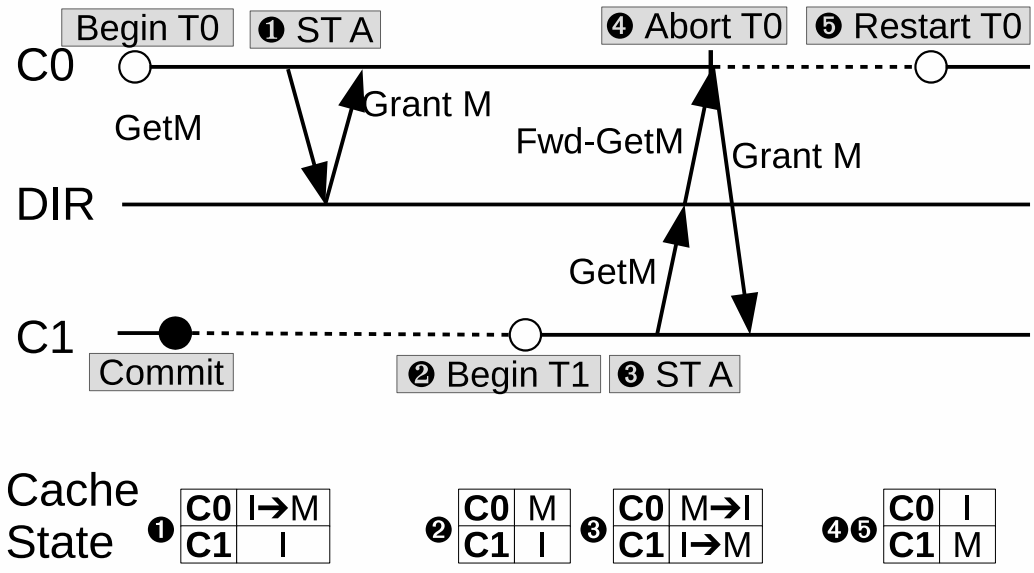



Figure 2.1: Timeline of wasted work in requester-wins HTM

At the beginning (left side of the figure), `C0` starts a long transaction `T0`, and `C1` has just committed a previous transaction. At ❶, `T0` does a speculative store to cache block `X` (all of our examples work at cache block granularity), so `C0` issues a `GetM` (fetch block

with permission to modify it) request. The directory finds that the block is not owned by anyone, and grants the permission to C0. Core C0 now has the block in M state, as shown in the Cache State part of the figure.

At ❷, C1 starts a new transaction T1. When T1 does a speculative store to address X at ❸, we have a store-store conflict that needs to be resolved. Since T0 and T1 may have performed other speculative memory accesses, we would violate atomicity if we allow both to proceed.

However, the unmodified coherence protocol is unaware of this transactional conflict. It implements the normal behavior for a GetM request received by a cache that has the block in the M state – C0 responds with the block's (original) data and invalidates its own cached version of the block. This response grants C1 the write permission it needs, so when it receives the response it goes ahead with the write. In this conflict, only C0 (the responder) is aware of the conflict. Because C1 will simply forge ahead when it gets the response, correct conflict resolution requires C0 to abort its transaction upon detecting the conflict. As a result, this type of HTM implements the "requester-wins" conflict resolution policy out of necessity.

However, the "requester-wins" decision is often a poor one. Transaction T0 is frequently the one that has done more work – after all, it accessed block X first. Even when the two conflicting transactions are both small, aborting C0 is often disadvantageous because, given a little more time (by delaying T1's write), T0 might have committed and no abort would have been needed.

Aborted transactions may exacerbate problems when they roll back and restart. If an aborted transaction restarts too soon, it may then re-request the block on which it previously had a conflict, and thus abort the winner of the previous conflict [1]. In the example in Figure 2.1, when T0 restarts at ❺, the hope is that, by the time T0 (re)reaches its instruction ST X, T1 will have already committed. If T1 has not committed by then, the HTM enters what can be a live-lock situation, where T0 and T1 repeatedly abort each other. This

problem is an example of the "Friendly Fire" pathology described in [11].

### 2.1.2 Prior Solutions

If we allow modification to the coherence mechanism, there are ways to reduce the impact of these issues. For example, in LogTM [12], transactions who access the line later (such as `T1`) are sent a negative acknowledgment (Nack) message. While `T1` is stalled, the earlier transaction (`T0`) is allowed to continue, hopefully committing before `T1` asks for the line again. Stiff-arming [3] is another mechanism to briefly stall requesters by holding on to the conflicting request at the directory in the hope that the earlier transaction manages to commit soon.

Instead of simply letting the transaction that accessed the line earlier stall the other transaction, FasTM [13] proposes a mechanism (called FasTM-Abort) that can instead abort the requester, or even decide between the transactions which one to abort. FasTM adds an additional `T` state to the MESI protocol, and allows cores executing a transaction to send a `Nack` message for blocks in the `T` state (like LogTM). The `Nack` message can optionally contain the timestamp when the transactions had started. The recipient of the `Nack` uses this information to figure out which transaction started earlier, and aborts itself if it's the younger transaction.

In HTMs with lazy conflict detection such as BulkTM [4], data conflicts are only detected at commit time. Among conflicting transactions, the one that reaches the commit point first wins, which ensures forward progress. However, here again we need to use TM-specific coherence.

Instead of trying to manage conflicts directly, best-effort HTMs can use fallback paths instead [14, 15, 16]. When a transaction is aborted, instead of simply trying again the abort handler can fall back to a mutex lock or other more sophisticated software work-arounds. Designers of the abort handler can also try to avoid conflicts more proactively by delaying a transaction that is expected to conflict with another one (transaction scheduling [17, 18,

19]).

However, abort handlers either act after the fact (when an abort has already occurred) or require good prediction of future aborts. Such prediction is difficult to do accurately, and sophisticated prediction code may result in overheads that negate much of the advantages gained by avoiding aborts.

## 2.2 PleaseTM

### 2.2.1 Overview

A key design constraint for best-effort HTMs is to avoid modifying the coherence protocol. The problem we are trying to solve is that this constraint seems to require that aborts are unavoidable when a transaction receives coherence invalidate/downgrade messages for transactionally-held data. Our solution is to provide a mechanism that decouples the coherence protocol from transaction conflict detection and resolution, which opens the door for using better conflict resolution policies.

The mechanism we propose is to extend coherence response messages with a plea bit (or bits). The coherence hardware ignores the plea bit(s) and simply passes them to the cache controller and the HTM engine of the requester. When a transaction receives a coherence protocol request that will cause it to abort, it can now use the plea bit to inform the requester that a conflict has been detected, and that proceeding with the requester's access will result in aborting another transaction. The requesting transaction, when it receives the plea, can elect to abort itself, thus resolving the conflict in favor of the pleading transaction[1]. We are in essence providing a conflict resolution mechanism similar to FasTM-Abort, but *without* any changes to the coherence protocol.

Because the coherence protocol is unmodified, the pleading transaction needs to invalidate/downgrade the conflicting block in its own cache before responding (with the plea).

[1]The requesting transaction may instead override the plea and continue, which will result in aborting the pleading transaction. We will investigate this later

Thus, after responding, the pleading core/cache must discover whether the data has been accessed by a third core in the mean time. The pleading core must re-acquire the invalidated/downgraded permission for the conflicting block and verify that the atomicity of its transaction is still intact.

Both goals are achieved by re-requesting the block, using `GetM` if the line was previously in the `M` state, or using `GetS` if it was in `E` or `S` state. The permission alone is not sufficient – the pleading core must check if the block was modified while it was "absent."

The solution we use is to validate the block's data when the block is re-acquired [20, 21]. Recall that HTMs with unmodified coherence protocols need to record the original data separate from the speculatively updated data. The original data is needed to rollback aborted transactions and respond to the abort-causing request. In PleaseTM, this original data is now used to verify the data of re-acquired cache blocks. If the data is unchanged, the work already done in the pleading transaction is still valid and the transaction may proceed. If the data validation does not succeed, transaction's work was based on data that is now stale, and the pleading transaction needs to abort.

Until we have confirmation that the plea was accepted, the transaction can continue executing, but is not allowed to commit. It the transaction reaches the commit point too early or receives additional memory requests to blocks pending refetch, it needs to stall until the block is re-acquired.

Despite a time gap, and possible intervening accesses, between the sending of the plea and data validation, the pleading transaction is still correct. The behavior of the transaction is as if the data it accessed and the work it did occurred atomically at the *serialization point* [10], when the transaction commits. Since we validated that the refetched data is unmodified, the speculative work done by the transaction is still valid, and thus may be committed. In addition, non-transactional accesses will ignore the plea bits and always win the conflict, providing *opacity* [10].

Figure 2.2 shows the architecture for PleaseTM, with the additional components shown

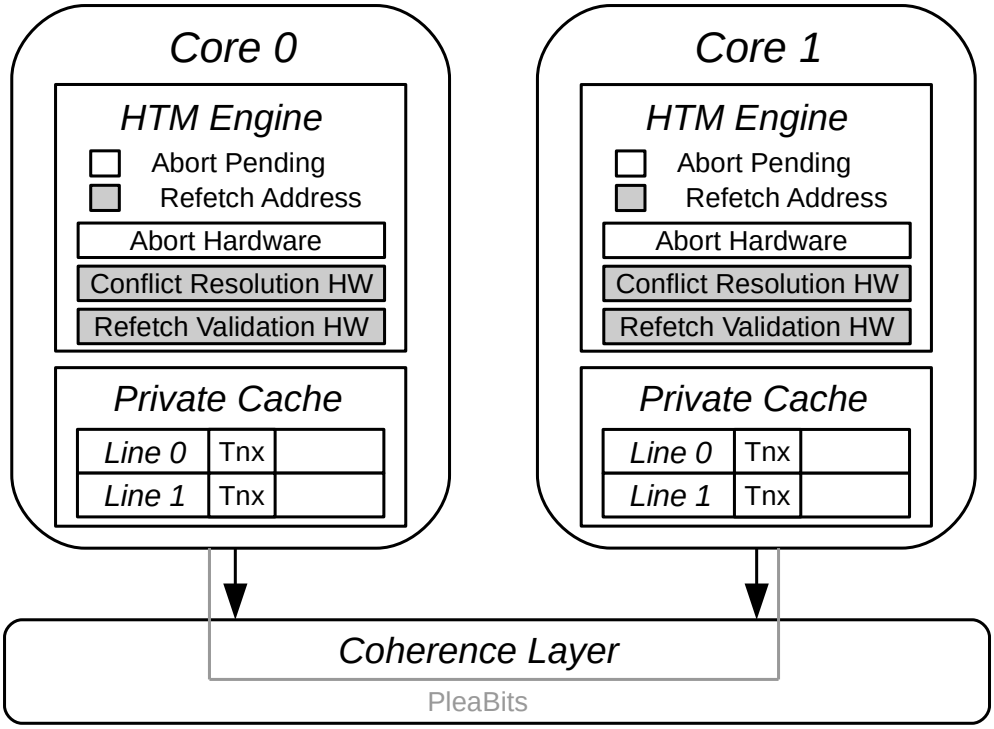



Figure 2.2: Architecture for PleaseTM

in gray. When a coherence request is received for a cache block with its transactional bit set, the *Refetch Address* register is set to the block's address. The responder core (or pleading core) will send a coherence response as usual, but with the plea bit set. This *Refetch Address* register is checked by the *HTM Engine* when trying to commit the transction, and if set will stall the commit until the refetch is complete.

The requester core's *Conflict Resolution Hardware* takes the plea and makes a local decision on whether to accept or override the plea. If the plea is to be accepted, the *Conflict Resolution Hardware* invokes the abort hardware to abort the transaction. If it is to be ignored, the execution continues as usual. If the requester is *not* inside a transaction, the *Conflict Resolution Hardware* is not active and the plea is simply ignored.

The pleading core later sends an additional coherence request to recover the line (either a `GetS` or `GetM`) to the requester. The *Refetch Validator* component will validate the re-acquired cache block to check whether the data has not changed since. If not, then the

*Refetch Address* is cleared and the transaction is free to continue. However, if the data *has* changed, this means that the speculative data the transaction was working with is now invalid, so the transaction has no choice but to abort.

The coherence layer does have one minor change required. The coherence layer needs to be augmented to include the additional plea bits in the coherence messages. However, these are treated as strictly payload and the coherence layer does not bother with it (and simply passes the information up to the conflict resolution layer).

We now describe in detail what needs to be done to handle various conflict resolution scenarios in PleaseTM.

### 2.2.2 Write-write Conflict with Plea Accepted

Returning to our example, Figure 2.3 shows what happens with our proposal. At ❷, transaction `T1` executes a store operation to block `X`. This request is forwarded to the current owner, which is `C0`. `C0` is executing `T0`, and wishes to continue with the transaction. When it responds to the coherence request, `T0` inserts the plea, which `T1` receives. The directory now indicates that `C1` is the new owner of the address. However, `T1` notices the plea bit sent along with the response, and therefore politely aborts itself (❸).

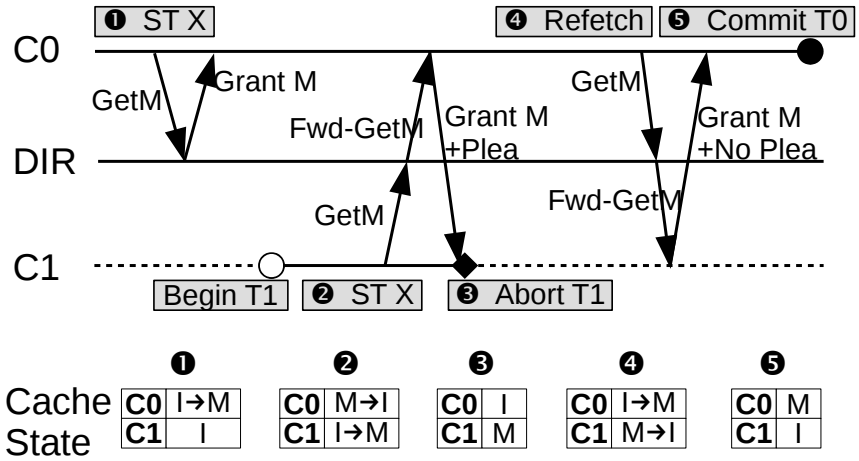



Figure 2.3: Write-write conflict with T0's plea accepted

Core `C0` must reacquire write permission for block A (and validate the data) before

continuing T0. After sending the response to ❸, T0 sends a GetM request (❹). The directory forwards the request to C1, which responds without any plea bits (since it has already aborted). When C0 receives the coherence response, it once again holds X in M state. Now T0 validates the data (successfully) and T0 continues with the transaction and later commits (❺).

### 2.2.3 Write-write Conflict with Plea Denied

In the previous example, T0 avoided an abort, but this won't always happen. Figure 2.4 shows T0 conflicting with non-transactional execution from C1. In this example C1 does a regular store operation at ❷ (recall that we work at cache line granularity, so this can happen when there is false sharing). Since C1 is *not* inside a transaction, it ignores the plea sent by T0 and updates the line (❸).

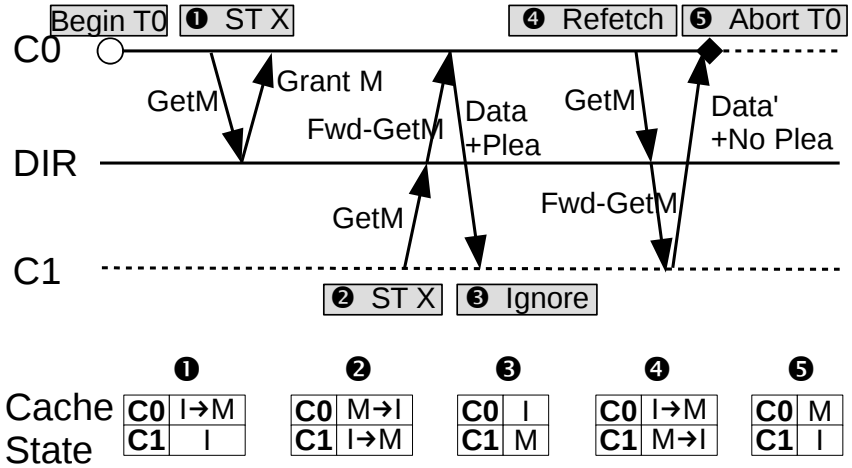



Figure 2.4: Write-write conflict with T0's plea denied

Transaction T0 does not yet know whether its plea was granted or not. It refetches the line (for validation) at ❹ through a GetM request. The directory forwards the request to C1, which returns the modified data (no plea bit here either, since C1 is not inside a transaction). C0 tries to validate the data, but this time the data does not match its buffered version. This indicates that there was an intervening write, that C0's copy of X is now stale. Even though C0 recovered the original permissions, C0 must abort T0 to guarantee atomicity, as shown at ❺.

### 2.2.4 Read-write Conflict with Plea Accepted

Figure 2.5 shows an example where C0 transactionally reads a line, and C1 later tries to do a transactional write to the same line. Like in the write-write case, we can use the plea bit to keep the earlier transaction from aborting. At ❶, C0 issues a GetS request corresponding to the load, and changes the line state to E or S state. Later, T1 starts, and C1 issues a GetM request (❷). When C0 receives the Inv message from the directory, it sends an Inv-Ack message back to C1. T0 inserts the plea bit in that message indicating that the line has been speculatively accessed by an active transaction that wishes to continue.

T1 receives the message and aborts (❸), discarding all speculative updates it had done. At ❹, C0 later refetches the line using a GetS message. C1, since its transaction has now aborted, returns the (unmodified) data without any plea bits set. T0 checks the data for validity, and since the data is untouched, proceeds to successfully commit (❺).

Even if there were multiple readers when C1 issued the GetM request, the end result will still be the same, albeit with pairwise plea bits and refetches between C1 and the readers. C1 sends an Inv message to the readers, and they all insert the plea bit into the Inv-Ack response. The first plea-enabled Inv-Ack response will abort T1, with the latter plea bits simply ignored (since C1 is no longer within a transaction). The readers each separately re-issue the GetS to C1 and continue with their transactions.

If C1 was *not* inside a transaction and did the write to cache block X, then the plea is

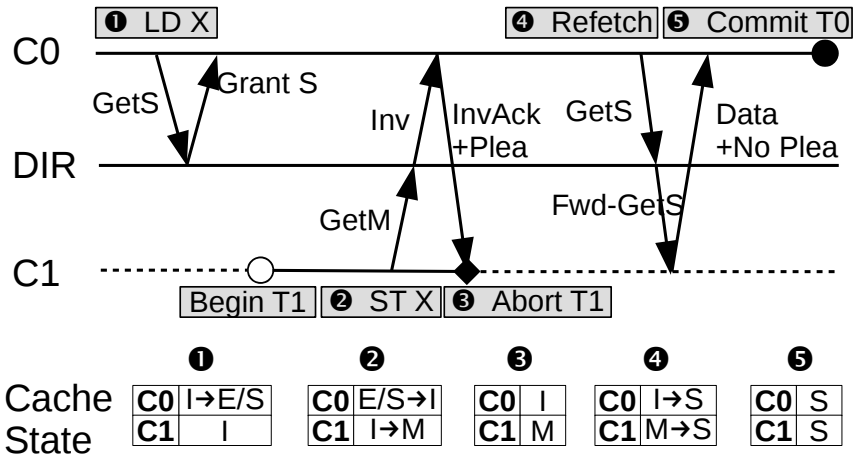



Figure 2.5: Read-write conflict with plea accepted

denied, just like in Section 2.2.3. Similarly, multiple readers are aborted after they refetch, as all their copies are now stale.

## 2.3 Additional Conflict Resolution Policies

In the previous section we discussed how adding a plea bit to coherence responses for a core with an active transaction can save longer-running transactions from aborts. By adding a simple plea bit to the coherence response and validating the data, we changed a requester-wins HTM to a requester-loses HTM. However, it has been shown [22, 13] that more sophisticated policies can do better than these simple policies. If the responding transaction `T0` has actually done less work than the requesting transaction `T1`, then it can be more beneficial to abort `T0` instead.

To realize such policies, the conflicting transactions need information about each other. For example, the timestamp when the transaction started, the number of aborts, or number of transactional reads or writes can all be used to estimate the amount of "work" each transaction has done so far. This information then needs to be communicated to the other transaction. In FasTM, the authors propose embedding this information in the `Nack` message.

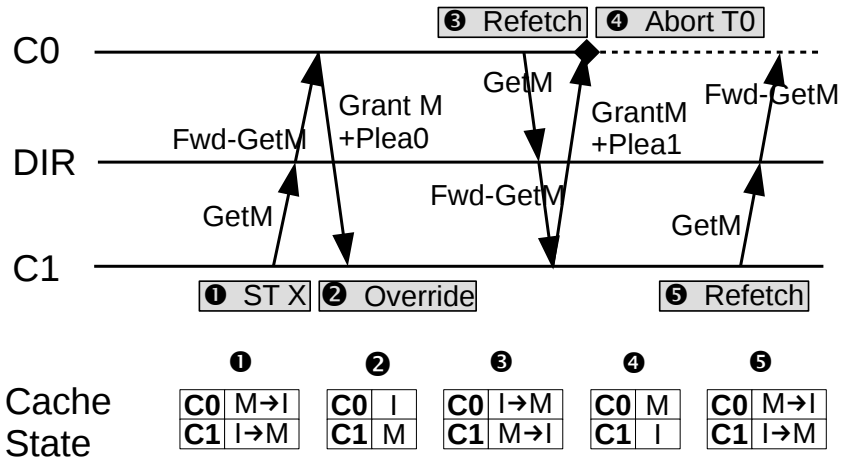



Figure 2.6: A write-write conflict with a different conflict resolution policy

In PleaseTM, we can expand the plea to include this information instead. Figure 2.6 shows how the system might use multiple bit pleas to support more sophisticated policies. Cores `C0` `C1` are each running transactions, and transaction `T0` has speculatively written to `X`. `T1` does a transactional store at ❶. `T0` receives the coherence request, and includes plea `Plea0` in the response. `Plea0` contains a number estimating the work done by `T0`. At ❷, `T1` receives this plea, and compares the included number with its own value. `T1` determines that it had done more work than `T0`, overrides the plea and continues instead of aborting.

Later, at ❸, `T0` refetches the line by sending a `GetM`. `C1` doesn't remember or care that `C0` previously owned the line. Instead, it tries to save `T1` by sending its own plea, `Plea1`. The plea comparison between `T0` and `T1` still results in `T1` winning, so this time `T0` accepts the plea and aborts (❹). At ❺, `T1` refetches the line, and validates the contents, hopefully committing at a later time.

Since refetch is not instantaneous, by the time the refetch operation arrives at `C0`, `T0` might have restarted and did more work than `T1`. In this case, `T0` will continue. This may lead to plea bits being sent back and forth between `C0` and `C1`. However, one of these transactions will either commit, or will eventually exceed the abort limit and invoke the fallback mechanism.

Since extra information takes up additional plea bits in each coherence response, we want to minimize its size. Most transactions are relatively small. Further, we are primarily interested in preventing a transaction that has done little work from aborting a transaction that has done a lot of work. Thus, we do not need a fully precise measure of the work done by a transaction. We investigate various possibilities for the information in Section 2.5.3.

## 2.4 Correctness

Our PleaseTM proposal makes no changes to the coherence protocol, and thus does not affect its correctness. Our hardware changes involve the cores sending additional coherence messages, but they are conventional messages. The plea bits are completely ignored by the coherence protocol and are simply passed up to the core.

Transactions may occasionally form cycles of aborts (livelock), but requester-wins HTMs do not guarantee the absence of such situations. For forward progress, they instead rely on fallback paths like global lock fallbacks. By better managing conflicts, PleaseTM can reduce such pathological aborts and rely less on those fallback paths.

Data validation is an important factor in ensuring correct behavior of transactions in PleaseTM. To avoid aborting, a responding transaction must successfully refetch data blocks in the original coherence state and with the original values.

In some cases, the pleading transaction may miss some write operations, but correct behavior is still ensured. The validation will not detect silent writes, where the write uses the same value that was already in that location. The transaction may have read the value prior to the silent write, but it acts as if it had read the data (whose value is unchanged) after the silent write. In other words, the *serialization point* is still at the point of the commit instruction.

The core still needs to respond to coherence requests while the refetch is outstanding. If an additional request does not indicate a conflict, we can just complete the refetch. If any additional requests indicate a conflict, it is theoretically possible to respond with more

pleas. However, this would require more extensive hardware support to track the multiple ongoing pleas and validate them all. We limit the complexity of our implementation by simply aborting the transaction if it receives a conflict-indicating coherence request when a plea is already outstanding. Note that the refetch to validate the original plea might have already been sent. Because the refetch is a normal coherence request, it will eventually complete normally even though the refetched coherence state is no longer needed (transaction had since aborted).

### 2.4.1 ABA Problem

We'll look at a more concrete example of a corner case that might occur under PleaseTM, and show why it does not violate correctness. The example is called the ABA problem, where data is overwritten, then overwritten back to the original value ("A → B → A"). This can occur because the refetch and validation of data blocks is not atomic. Following the same reasoning as for silent writes, this does not violate correctness. Since the serialization point is the transaction commit, what's important is what the value was at that point: all memory operations appear as they all performed instantaneously: the transaction did work after reading A.

Figure 2.7 depicts a sequence of events that represents this problem. There are three cores now; core `C0` is running a transaction (`T0`), while core `C1` and core `C2` is running outside of a transaction. In addition to the message diagram and cache state transitions, we also show the value of the data block in question, `X` in the bottom row.

At time ❶, `T0` speculatively accesses data block `X`, reads A and speculatively updates it to S. Later, at time ❷, core `C1` does a non-transactional store to the same block `X` to the value of B. Since core `C1` is outside of a transaction, it ignores the plea bit sent by core `C0` (but core `C0` hasn't been notified of this yet). At time ❸, another core `C2` writes to `X`, turning it back to A.

At time ❹, core `C0` attempts to refetch data block `X`. It sends a `GetM` message, which

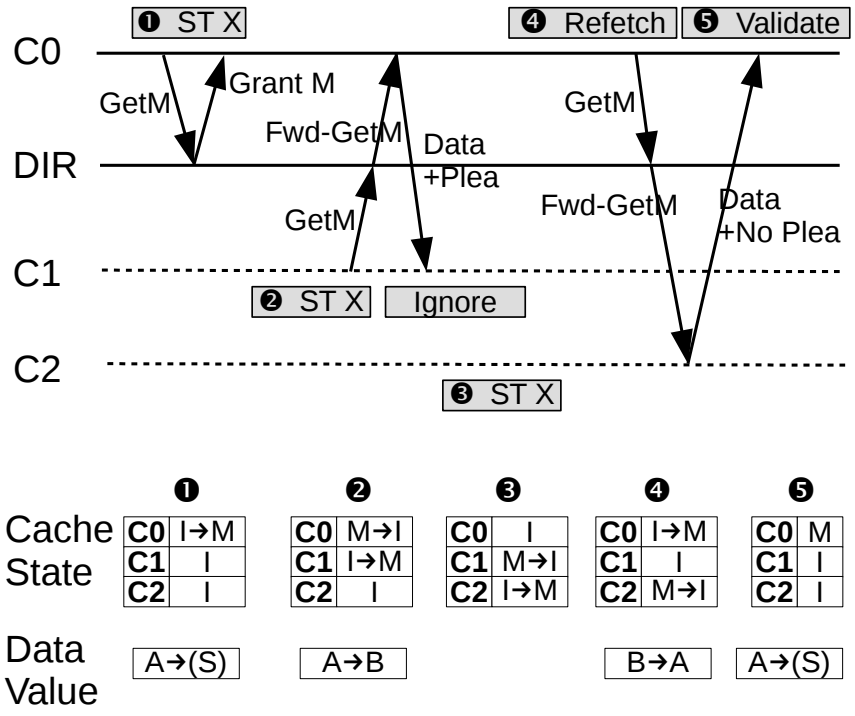



Figure 2.7: Timeline for A → B → A

is forwarded to core `C2`. Since the value of `X` is A at this point, `T0` validates the refetched data, and proceeds as usual. This is correct because we are ordering the writes by core `C1` and `C2` *before* the commit of transaction `T0`.

## 2.5 Experimental Results

### 2.5.1 Setup

We modify SuperTrans [23], a transactional memory simulator built from SESC [24], to more accurately simulate best-effort HTM similar to Haswell's Transactional Synchronization Extensions (TSX). The L1 cache stores speculative (transactional) data and the L2 cache maintains the original values. Conflicts are detected when coherence invalidations/-downgrades are received for transactional data. Replacing a transactional dirty line results in aborting the transaction. Replacing clean transactional data puts the block's address in an overflow set, and triggers an abort only if the overflow set is already full. Non-transactional

lines do not affect the transaction when replaced. Inclusion is enforced between L1 and L2, so L1 replacements (and transaction aborts) can be caused by L2 replacements.

Each core has private 32KB L1 instruction and data caches, and a private 256KB L2 cache. The L1 caches are 8-way set associative with hit latency of 3 cycles, and the L2 caches are 16-way with 20 cycle hit latency. On transaction abort, speculatively written lines in the L1 are invalidated, reverting to the version in the L2. On commit, these are written back to the L2. Invalidation of any speculatively accessed line causes an active transaction to abort.

We model 36 cores that are connected through a 6 by 6 mesh network with X-Y routing. Coherence among private caches is maintained using a directory-based MESI protocol. The shared L3 cache is distributed, with a 1MB slice in each of the 36 tiles, and the off-chip memory has a 250-cycle latency. The threads are pinned to each core, so that no core runs with more than 1 thread. Execution time (and speedups) are based on the "Time =" output in each application. The simulations are run with GNU parallel [25].

We evaluate the benefits of our approach using STAMP benchmarks [26] with recommended simulator inputs (excluding the bayes benchmark, which has a non-deterministic runtime making it unsuitable for performance comparison). Kmeans and vacation have two input sets, and we denote the results from high-conflict input with a "+" suffix. Since we model a best-effort HTM, we need an abort handler to guarantee forward progress. Our default handler uses a randomized linear backoff between an abort and a retry. After a number of repeated data conflict aborts (the actual threshold used is the best performing one across all benchmarks for each policy), the handler takes the fallback path which acquires a global lock instead of starting the transaction [14].

### 2.5.2 Overall Results

Figure 2.8 shows the parallel speedup (relative to the single-threaded run) for several conflict-resolution policies. The first policy shown is the baseline *RequesterWins* policy

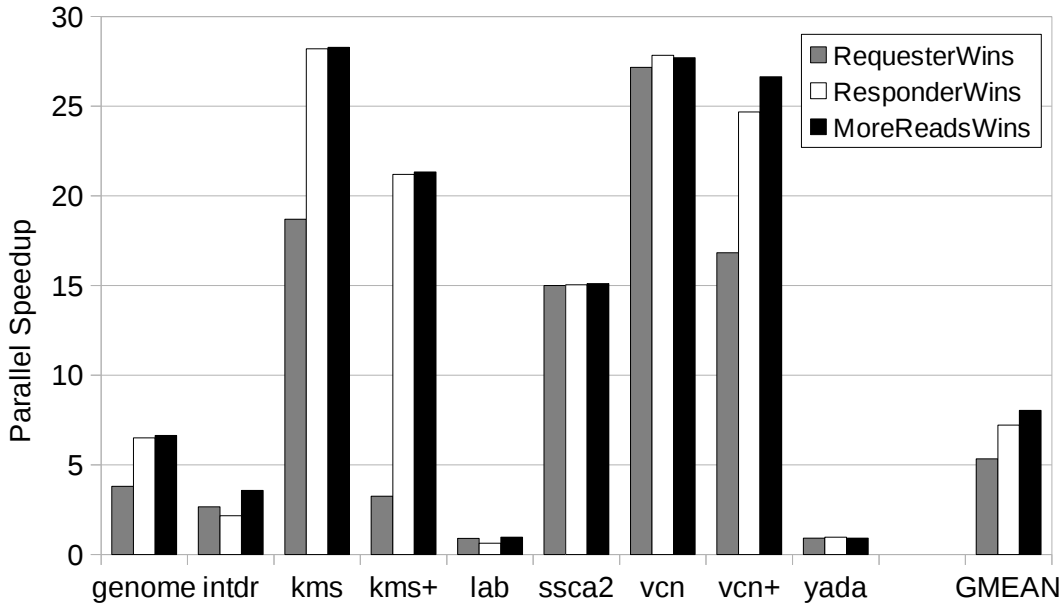



Figure 2.8: Baseline (*RequesterWins*) vs. Two Simple Plea-enabled Policies

used in best-effort HTMs, where the transaction requesting the block "wins," and the transaction(s) that receives the invalidation/downgrade request loses (is aborted).

The other two policies are enabled by our plea-based approach. In *ResponderWins*, a transaction that receives an invalidation/downgrade for a speculatively accessed line does not immediately abort, but sends a single-bit plea with the response. When the requester core receives such a response, if it is executing a transaction, it aborts. In short, this policy resolves conflicts in the exact opposite way from the baseline *RequesterWins* approach.

In the *MoreReadsWins* policy, the responding transaction sends a plea that includes a few bits about how many blocks it has read transactionally. The requesting core, if executing a transaction, compares this to its own number of transactional reads and aborts its own transaction if it has fewer reads so far. Otherwise, it ignores the plea and continues executing the transaction, causing the responder to eventually abort when its refetch fails. In short, this policy resolves conflicts by aborting the transaction that has done less "work," for which we use the number of lines read as a proxy.

Overall, we observe that plea-enabled policies tend to outperform the *RequesterWins*

baseline, especially in applications whose performance scales well (kmeans and vacation). The improvement is primarily due to resolving conflicts such that it aborts the transaction that has done less work, and preserves the one that has done more work. Although *ResponderWins* does not explicitly measure or compare the work done by conflicting transactions, it aborts the requester, which tends to be the transaction that did less work. On the other hand, *MoreReadsWins* explicitly prefers the transaction that did more work. In genome, intruder, labyrinth, and yada, the information on which transaction should be kept alive provided by *MoreReadsWins* gives it an additional benefit above and beyond *ResponderWins* – whereas *ResponderWins* always aborts the transaction that receives the plea, the *MoreReadsWins* policy benefits from being able to do the opposite when the responding transaction is actually the "smaller" one. Intruder and labyrinth are interesting in that the *ResponderWins* policy is inferior to the *RequesterWins* baseline, whereas *MoreReadsWins* makes up the lost ground and even improves upon *RequesterWins*. Finally, ssca2 experiences so few data conflicts that conflict resolution doesn't make much of a difference.

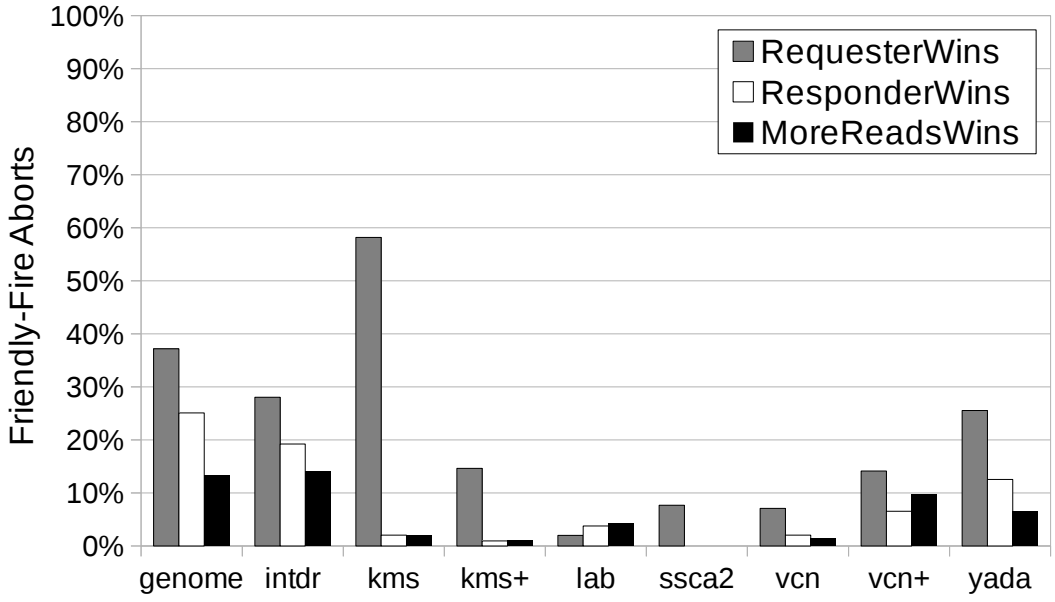



Figure 2.9: Ratio of Friendly-fire Aborts

Figure 2.9 shows the percentage of friendly-fire aborts [11] from the total number of aborts each benchmark experiences. Friendly-fire aborts are defined as aborted transactions

who themselves have aborted another. As explained in Section 2.1, when using the baseline *RequesterWins* policy, and the aborted transaction restarts too soon, the system can get caught in a situation where the aborted transaction restarts and aborts the aborter, which itself then turns around and aborts the first transaction again, resulting in a live-lock.

We can see that with better conflict resolution, we can significantly reduce the ratio of friendly-fire aborts in most of the benchmarks. The effect is most pronounced in kmeans, where a simple flipping of the abort decision outcome removed almost all friendly-fire aborts. This is because conflicts are detected early in the transaction, so once a transaction starts "winning," it becomes very likely to commit successfully. In vacation, we do see a slight increase with *MoreReadsWins* compared to *ResponderWins*, but this is compensated by the reduction in the total number of aborts.

In summary, our proposal to add a plea to coherence responses enables better conflict resolution policies that significantly improve performance. *MoreReadsWins* appears to be a good policy, but even better and/or cheaper policies might be possible; the key takeaway is that *the plea enables improved policies*. Put another way, our decision-enabling approach is important for improving the performance of best-effort HTMs. The baseline *RequesterWins* policy, the only choice in today's best-effort HTM systems, tends to force poor conflict resolution decisions, so enabling even a simple reversal of these decisions (*ResponderWins*) tends to be beneficial. Our plea-based approach allows more sophisticated (and better-performing) policies to be implemented, as exemplified by the *MoreReadsWins* results. Even more sophisticated policies, such as adaptive policies informed by prior behavior, may do even better, and such policies are only possible for best-effort HTMs once choice is enabled by a mechanism such as ours.

### 2.5.3 More Conflict Resolution Policies

As shown previously [22, 13], using more sophisticated conflict resolution policies can result in better performance. By utilizing the plea bits, PleaseTM can also support better

conflict resolution decisions by embedding the information in plea bits. We've already shown that the *MoreReadsWins* policy has a significant advantage over the simple 1-bit plea in *ResponderWins*.

Other than these two policies, we also consider the following policies. The *OlderWins* policy gives higher priority to transactions that have executed more instructions. It uses the plea bits to hold the number of instructions the active transaction has executed since it started or restarted. The *WriterWins* policy is has the writer always win the conflict. The *MoreAbortsWins* policy gives higher priority to transactions that have experienced more aborts. It uses a plea containing the number of aborts the active transaction has experienced so far.

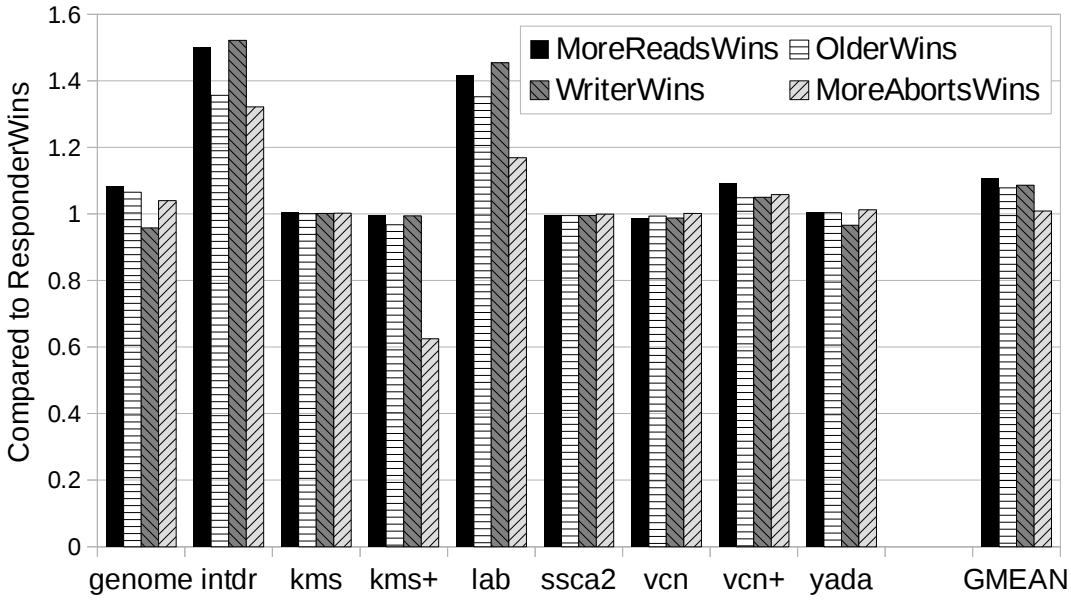



Figure 2.10: Speedup of Various Types of Plea Bits Compared to *ResponderWins*

Figure 2.10 compares the performance of the different policies, normalized to *ResponderWins*. *OlderWins* performs similar to *MoreReadsWins*. This is because the two metrics of work are highly correlated; executing more instructions generally means executing more reads. This is especially true when the two transactions share the same static code (i.e., if the threads were using locks, they would be in the same critical section). While *OlderWins* treats all instructions within a transaction as equivalent, in *MoreReadsWins* only reads have

weight. The idea behind *MoreReadsWins* is that speculative accesses are the most important measure of work, since they expose the transaction to conflicts. Also, *MoreReadsWins* can theoretically use a more compact plea, since not all instructions are reads.

*WriterWins* largely performs similar to *MoreReadsWins*, but performs worse in genome. This is because we always favor the writer, even if the requester sends a `GetM` request and conflicts with multiple readers; each of which might have done more work.

*MoreAbortsWins* performs worse than the other policies, and with kmeans, even worse than *ResponderWins*. The reason for this is twofold. First, our use of a global lock fallback means that, when it is used, all transactions except the one falling back get aborted; this can skew the notion of work done by the transactions. Second, the granularity of this metric is too coarse, leading to many transactions having identical plea values.

### 2.5.4 Overhead of Avoiding Coherence Protocol Modifications

Compared to FasTM-Abort, conflicts in PleaseTM can generate additional traffic due to refetch. In FasTM, when there is a coherence request to a conflicting data block, transactions can respond with a `Nack` message that includes information about the responder transaction. However, instead of the single `Nack` message required to abort the younger requester, in PleaseTM the responder transaction needs to send a response (embedded with the plea bits) and later refetch the data by sending a `GetS` or `GetM` message.

We next look at the overhead due to these additional messages. Figure 2.11 compares the performance of PleaseTM and (coherence modifying) FasTM-Abort. We can see that although there is a slight performance gap between FasTM-Abort and PleaseTM, it is small compared to the gap between *RequesterWins* and *MoreReadsWins*. Figure 2.12 shows the number of overhead messages as a percentage of all coherence messages in the system. An overhead message is either the refetch request and response message in the case for PleaseTM, or the `Nack` message in the case for FasTM.

We can see that the number of additional messages sent due to refetch (or `Nack`) is very

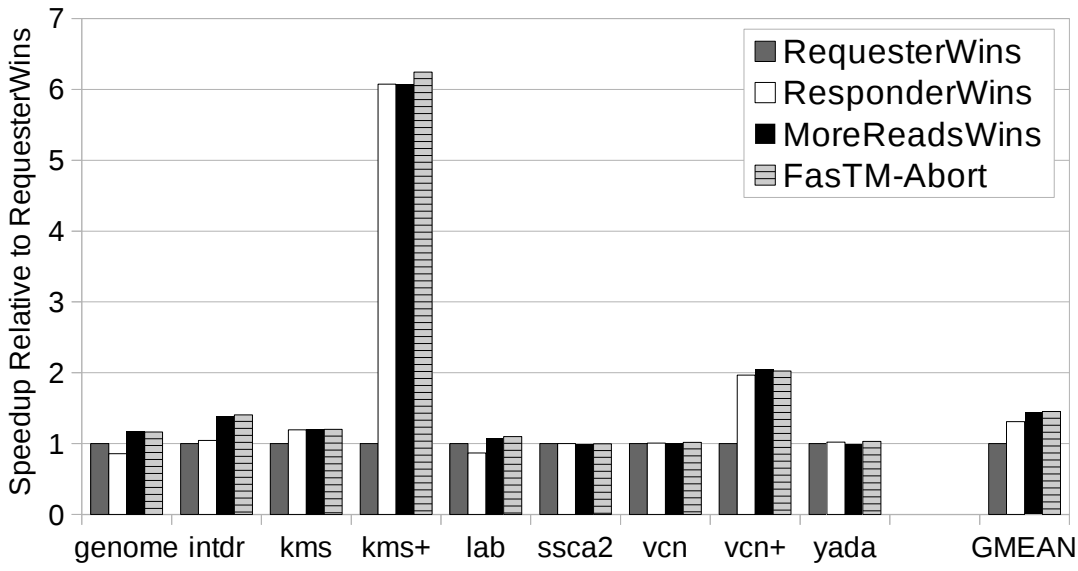



Figure 2.11: Performance Overhead of Avoiding Coherence Modifications

modest. Recall that these additional messages are only sent when transactions access data in a conflicting manner. Most of the data are to private data or data unique to that transaction. In addition, recall that in *RequesterWins*, the conflict always results in an abort. The aborted transaction will need to redo the work, possibly leading to more messages. This means that the additional messages due to PleaseTM are not always detrimental. Instead, by reducing the amount of lost work, proper conflict resolution often compensates for these additional messages.

### 2.5.5 Effect on Software Fallback Thresholds

When a transaction experiences repeated aborts, this can imply that the current part of the application does not have enough parallelism. When a thread experiences more than a given number of repeated aborts (the threshold), one way out of this situation is to take a software fallback path that acquires a global lock [14].

Because atomicity needs to be ensured between threads inside a transaction and the thread taking the fallback path, transactions need to keep the global lock in their read sets. Acquiring the lock will write to it, forcing all active transctions to abort, and allowing the

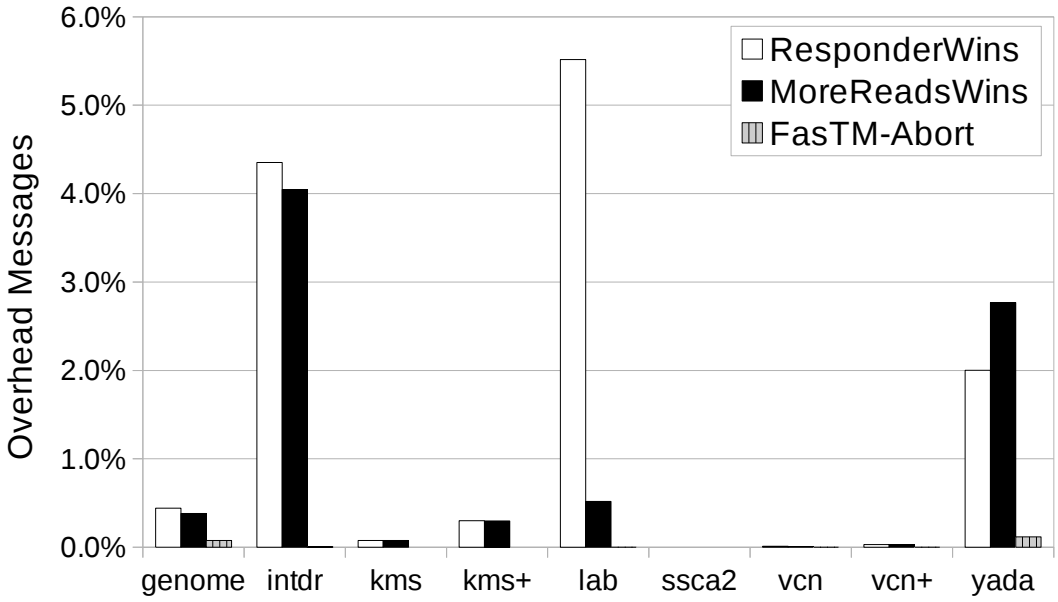



Figure 2.12: Overhead Messages (Refetch/Nack)

critical section to execute in isolation. Aborting all these transactions can have a performance impact.

By using PleaseTM with *MoreReadsWins*, and to a lesser extent with *ResponderWins*, we favor transactions that did more work. This means that the conflict resolution results will not change until one of them completes (unlike in *RequesterWins*, which depends on when the cache block was accessed, and is less deterministic). In other words, transactions that "lost" will continue losing. When the threshold is set too low, the losing transactions will quickly ratchet up their abort count. This will cause them to prematurely switch over to the software fallback path, often before the winner transaction has a chance to complete.

In figure 2.13, we compare the mean speedup of each conflict resolution policy, using fallback thresholds of 8 through 24, to the mean speedup when using the baseline *RequesterWins* with a threshold of 8. We can see that when using these policies, it is beneficial to have a high threshold value for triggering the software fallback.

Another takeaway is that because restarted transactions can themselves abort their aborter, it can be tricky to tune the baseline policy for better performance [8, 27]. With the threshold being too large, conflicting transactions may simply continue to abort each other

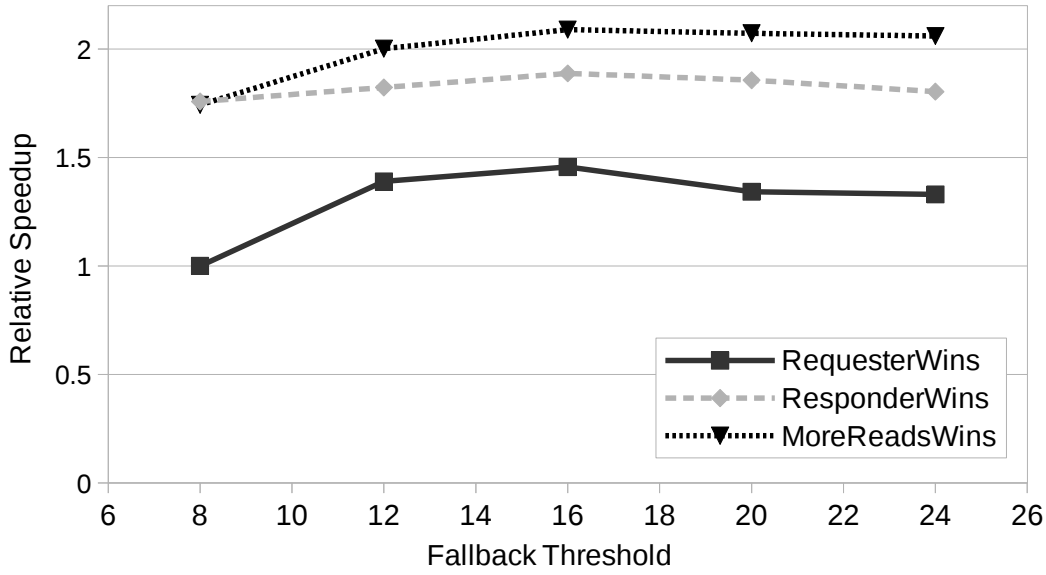



Figure 2.13: Speedup With Different Fallback Thresholds

without increasing the probability of success. However, as we can see from figure 2.9, *MoreReadsWins* settles on a certain "winner", reducing the ratio of friendly-fire aborts. With fewer friendly-fire aborts, the performance is more stable and thus less sensitive to the threshold value, reducing the burden of finding the "right balance."

### 2.5.6 Conflict Resolution and Software Fallback

Our baseline system uses a global lock fallback, as described in Section 2.5.1. While using a global lock is simple and straightforward, each use of the global lock can be detrimental to performance. When the global lock is acquired by someone, this causes all active transactions to abort, as described earlier.

This performance issue has motivated researchers to explore alternative software fallbacks. The fallbacks primarily decide if and when to retry a transaction, and thus treat the problem as a transaction scheduling problem ("when should I start this transaction?"). Conflict resolution policies such as ours are separate from transaction scheduling policies, but the two can work together [28]. We now explore how our plea mechanism, and the various resolution policies we support, interact with these alternative fallbacks.

One fallback proposal uses a token called the "hourglass" to limit the number of outstanding transactions without aborting everything [15]. When a transaction experiences more than a given number of aborts, instead of grabbing a global lock, it atomically sets a flag (the hourglass) and starts the transaction (needed to detect data conflicts). Once the hourglass is set, new transactions are not allowed to start (or restart). This is similar to the global lock fallback. However, unlike the global lock fallback, the hourglass holder does not abort active transactions. Instead, by blocking starts and restarts from other threads until the hourglass holder commits, transactions gradually drain out of the system.

Another proposal, called "Serialize-on-Killer" (SOK), tries to serialize conflicting transactions [29]. All threads have associated flags that are set before starting a transaction, and cleared after the transaction commits. When a transaction aborts, the hardware passes the ID of the aborter ("killer") thread to its handler. The software fallback uses this to busy-wait on the aborter thread's flag, which a thread clears when it commits a transaction.

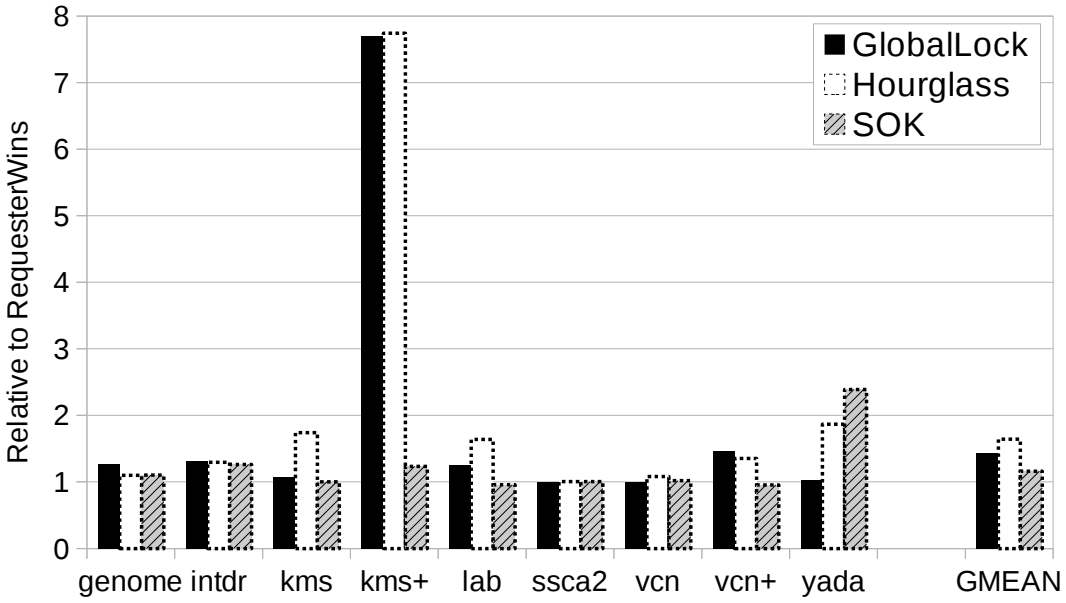



Figure 2.14: Speedup Chart Various Software Fallbacks

Figure 2.14 shows the speedup of *MoreReadsWins* over *RequesterWins* with different fallbacks. Each bar shows the speedup of *MoreReadsWins* over *RequesterWins* for that benchmark and fallback (in other words, each "GlobalLock" bar shows the speedup of

PleaseTM-enabled *MoreReadsWins* over the *RequesterWins* baseline, where both PleaseTM and the baseline use the GlobalLock fallback path. Likewise for the "Hourglass" bars and the "SOK" bars).

The benefits of a better conflict resolution policy actually improve with the more sophisticated Hourglass policy. The SOK fallback, on the other hand, is more mixed. Since aborted transactions are stalled until the aborter transaction commits, we see fewer friendly-fire aborts, and as a result less of a benefit to using *MoreReadsWins*.

### 2.5.7 Conflict Resolution and Software Optimizations

Another approach developers can take to improve performance when using transactional memory is by revising the software to experience fewer data conflicts. Nakaike et al. [30] analyzed the STAMP benchmarks and proposed several software changes. Genome is modified to use shorter transactions, while in kmeans care was taken to make the data align on cache lines better, reducing potential for false sharing. Intruder and vacation were modified to use data structures more suitable for HTM.

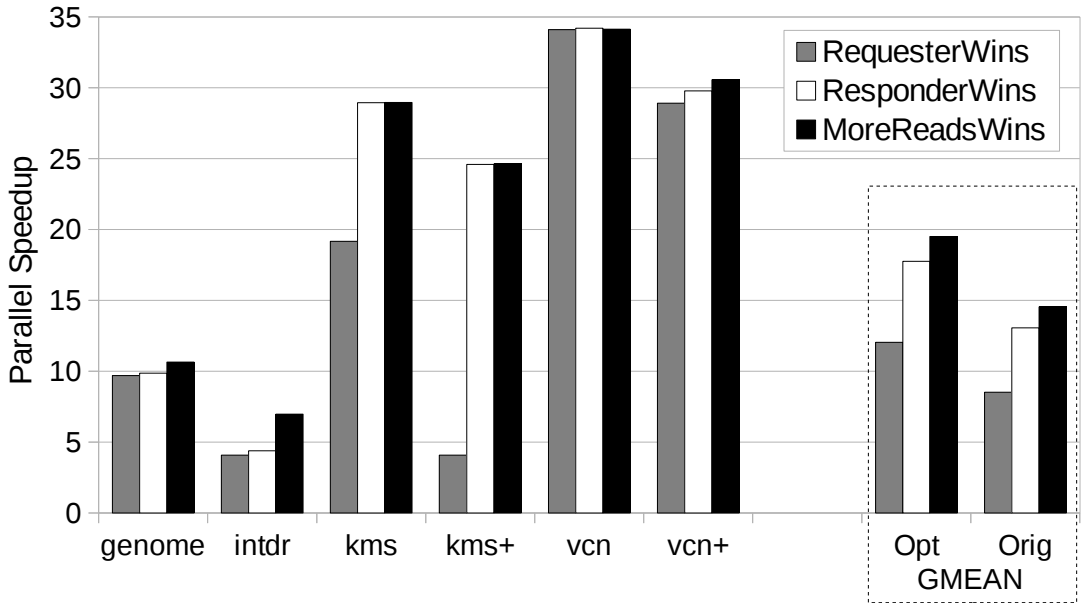



Figure 2.15: Speedup Chart with Optimized Software

Figure 2.15 shows the parallel speedup of each modified benchmark. On the far right, it also shows the mean as "Opt", and as reference, the mean speedup of the original code as "Orig." We can see that the software changes do improve overall speedups, but better conflict resolution policies enabled by PleaseTM still provide an additional benefit over *RequesterWins*.

## 2.6 Related Work

LogTM is an HTM scheme that detects data conflicts using a modified directory-based coherence protocol [12]. Like our scheme, LogTM attempts to keep older transactions from getting aborted by younger ones. However, unlike PleaseTM, LogTM does this by extending the coherence protocol (transactional values are immediately updates, with the original values in an undo log). When a transaction requests data that is currently held by another transaction, it sends a negative acknowledgment message (Nacks) to stall the requester.

FasTM [13] removes the software undo log requirement and simplifies the changes needed by LogTM, but still requires the `Nack` messages and acknowledgments. In addition, instead of stalling nacked transactions, FasTM has the ability to abort them (FasTM-Abort). By inserting timestamps in the `Nack` messages, FasTM allows longer-running transactions to abort shorter ones on a conflict. Akpinar et al. [22] and Shriraman et al. [28] use a similar mechanism to FasTM-Abort and investigate a variety of conflict resolution policies.

Constrained transactions in System Z assure forward progress for a small subset of transactions that comply with certain restrictions [3]. In addition, System Z also provides "stiff-arming" that allows transactions to temporarily stall an access. However, this is done through modifying the coherence protocol, complicating verification.

Instead of modifying the coherence protocol, there has also been work on separating conflict detection from the coherence protocol. Transactional conflict decoupling [31, 21]

predicts that invalidated cache lines result from false conflicts, and uses the stale values as if they were still valid. This allows speculative execution to continue without aborting, but requires a refetch and validation similar to our proposal. Suppressing silent stores [32] instead tries to eliminate unnecessary aborts from silent writes. When a core writes a value to shared memory that is the same as the old value, then the transaction keeps the data in shared state and avoids broadcasting coherence messages.

After the release of various commercial HTM systems, there have been several pieces of work on how to take advantage of these best-effort style HTMs. Yoo et al. [8] investigate the performance Intel's best-effort HTM system and briefly discuss how to tune for performance. Diegues and Romano [27] look at tuning parameters such as the fallback threshold at runtime using machine learning. Diegues et al. [33] discuss the challenges involved in extracting the best performance from transactional memory in general, and find that HTMs perform very well for certain transaction types (e.g., shorter ones), whereas fine-grained locking and software TM systems perform better in others.

After a transaction aborts, restarting too soon can lead to more aborts, especially in best-effort HTMs. Many schemes have been proposed to make better decisions on when to schedule the restart of a transaction. Adaptive Transaction Scheduling [16] takes a feedback driven approach and forces transactions into a run queue when the conflict intensity is too high. The run queue limits the number of active transactions, reducing conflicts. Toxic Transactions [15] proposes a software mechanism called the hourglass to throttle starting of new transactions when conflicts are frequent, but allowing those already active to complete. This is the same hourglass token used in Section 2.5.6. Software-assisted Conflict Management [34] is a contention management scheme that uses an auxiliary lock acquired by transactions that experience conflicts, forcing them into a run queue, while leaving non-conflicting transactions alone.

Instead of globally adjusting the number of active transactions, there are also proposals to predict likely conflicts and schedule transactions around them. Armejach et al. [29] look

at several proposals that serialize transactions so that restarted transactions happen one after another. Proactive Transaction Scheduling [17] uses the past history of transactions and the threads that execute the transaction to predict potential data conflicts and schedules transactions accordingly. Bloom Filter Guided Transaction Scheduling [18] uses Bloom filters of the read and write sets to inform the conflict predictor. Preventing versus Curing [19] instead uses the past history of the read and write operations to predict the transactions' access set to predict conflicts.

# CHAPTER 3
# TRANSACTIONAL PRE-ABORT HANDLERS IN HARDWARE TRANSACTIONAL MEMORY

## 3.1 Background

In a transaction, a thread optimistically works on shared data while checking for conflicts that (may) violate the transaction's atomicity. The TM system keeps track of which data each transaction has accessed: the read set and write set. Using these sets, transactions identify any potential read-write or write-write conflicts, and abort one (or more) of the conflicting transactions when a conflict is detected.

A transaction abort involves discarding its speculative state, placing some information about the cause of the abort in a status register, and then passing control to an *abort handler*. This handler chooses the desired course of action, e.g. retry the transaction immediately, retry after a brief wait, or use the fallback path.

The fallback path is an alternative to retrying the transaction, and is needed because commercial HTM implementations are "best-effort," i.e. they do not in general guarantee forward progress of transactions. For example, a transaction whose working set is larger than the private storage will be aborted on every retry, so the abort handler must eventually use the fallback path for such a transaction. A typical fallback path acquires a mutex lock, the fallback lock. The thread then executes the same code as the transaction's body, but non-speculatively, as a critical section.

The life a typical transaction is shown in Figure 3.1. First, it checks the status of the fallback lock (❶). This must be done *inside* the transaction to eliminate the race between the check and the start of the transaction, and to add the fallback lock to the transaction's read set. This ensures that the transaction will detect a conflict (and be aborted) if another

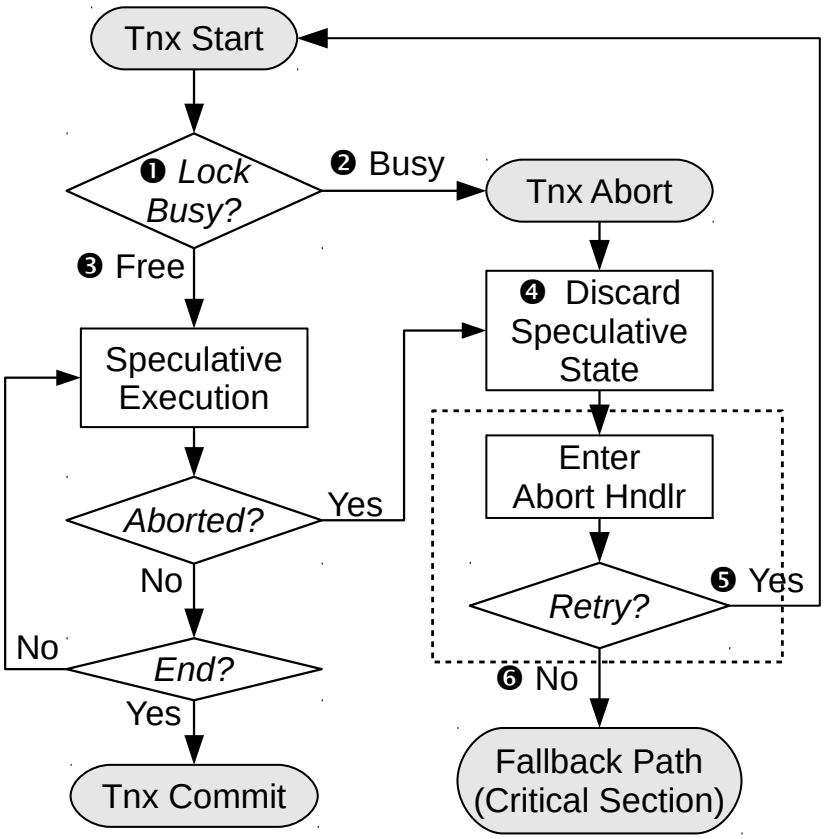



Figure 3.1: Life of a Transaction with the Baseline Fallback

thread acquires (writes to) the lock.

A busy lock (❷) means that another thread is using the fallback path, so the transaction aborts itself to ensure mutual exclusion for that other thread. Otherwise (lock is free, ❸), the transaction enters the actual body of the atomic section.

The transaction is executed speculatively by the TM hardware while checking for aborts. If it completes all of its work, the thread executes the commit operation/instruction, causing the TM hardware to atomically write to non-speculative state all of the data that was speculatively written. If an abort does occur, the speculative state is *automatically* discarded (❹) and the thread switches control to the abort handler, which can retry the transaction (❺) or use the fallback path for it (❻).

## 3.2 Pre-abort Handlers

In Figure 3.1, note that the speculative state is automatically discarded when an abort condition is met. For aborts caused by data conflicts, this is a reasonable approach. Data conflicts can lead to partial work being exposed and atomicity being violated. However, in many cases, the transaction is aborted because it cannot continue, discarding the work done so far even though that work is *still valid*. For example, a transaction that overflows the speculative buffer cannot continue because, if a data conflict does occur later on, the transaction's writes can no longer be rolled back.

This does not mean the work must necessarily be discarded. For example, when a transaction overflows the speculative buffer, the other threads could be paused by a hardware mechanism [35] to allow the overflowing transaction to complete correctly. The same effect can be achieved by atomically *converting* the transaction into an irrevocable transaction just before it overflows the speculative buffer. However, it is difficult to do so by anticipating the overflow in the transaction's own code [36]. Reactive approaches (conversion when buffer overflow is just about to occur) are precluded by the TM hardware automatically aborting the transaction without giving the software any opportunity to remedy the situation and possibly avoid the abort.

As a solution, we propose to allow the hardware TM to invoke a function, a *pre-abort handler*, when the transaction encounters an abort-causing condition (in other words, before ❹ in Figure 3.1). This will allow the programmer to decide whether to take mitigating action to avoid the abort-causing situation, or keep the default behavior and discard the speculative work.

Figure 3.2 depicts a high level view of the steps taken. Before we start the transaction, the entry point of the handler function is registered to the HTM system (①). Later, when the transaction encounters a situation that can trigger an abort (②), say a capacity overflow, we treat it as if the instruction had an exception; the instruction is quashed and the PC value

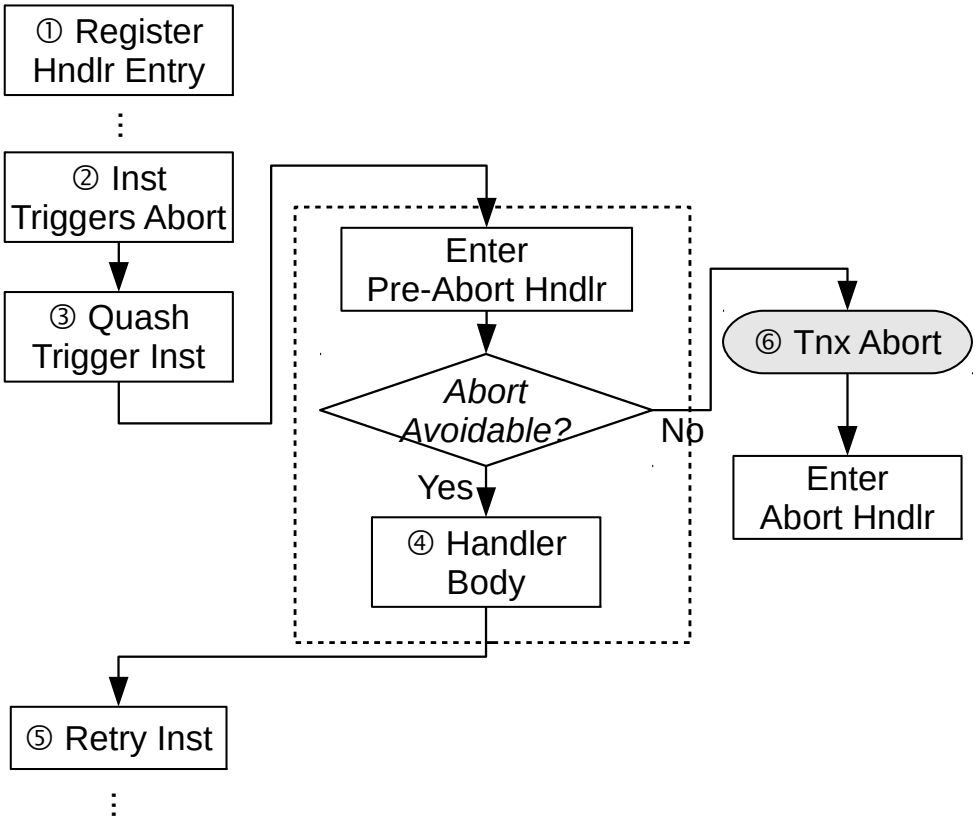



Figure 3.2: Flowchart With Pre-abort Handler

of the *trigger operation* is saved (③). Note that unlike typical exceptions or interrupts on processors that support HTM, this exception does not itself trigger an abort.

The thread is then diverted to the handler entry point. The handler is provided arguments such as the type of abort-causing situation, which can be used to decide which mitigating action to take (④). After the situation has been resolved, the handler returns. The PC of the trigger operation is restored, allowing the thread to re-attempt the operation (⑤). Hopefully, the abort-causing condition has been fully resolved, and the transaction is no longer threatened.

In some cases, the situation is unresolvable and the transaction has no other option but to abort. In this case, the handler code calls abort explicitly (⑥), causing the thread to follow standard abort handling procedure (i.e. abort and rollback the transaction, then enter the abort handler).

The fact that the pre-abort handler was triggered does *not* mean that the transaction is immune from abort. Other abort-causing conditions, such as regular data conflicts, are still tracked. If there is an additional abort causing condition during execution of the pre-abort handler (*nested aborts*), the transaction is aborted and speculative state discarded, with the program control passed directly to the (regular) abort handler. Pre-abort handlers must be carefully written, since they execute non-speculatively, but may be interrupted (and never continued) at any time.

### 3.2.1 Supported Abort Types

Table 3.1 shows what abort types we handle, and what additional arguments are passed (if any).

Table 3.1: Abort Types with Pre-abort Handlers

| **Cause** | **Handled?** | **Additional Arg** |
|---|---|---|
| Syscall | Yes | Number |
| Capacity | Yes | None |
| Manual Abort | Yes | Abort Argument |
| Conflict on Clean | Yes | Conflict Address |
| Conflict on Dirty | No | N/A |

When a system call is attempted we call the pre-abort handler instead of aborting. The handler is passed `syscall` as the cause argument, and an additional argument, the system call number.

When a transaction hits its capacity limit, the handler is called with the cause argument as `capacity`.

Many HTMs allow transactions to be aborted explicitly, e.g. by executing an XABORT or TABORT instruction [2, 1], along with a user-defined abort-cause argument. We also call the pre-abort handler in this case. The user-provided argument is passed as the additional pre-abort argument. In this case, the handler is set to return to the instruction after the abort

instruction (as opposed to returning to the instruction itself), since it makes no sense to retry the instruction and return right back to the handler.

The pre-abort handler is also invoked for conflicts on lines that were only read by a transaction handler (transactional and clean). In this case, the virtual address of the cache block that experienced the conflict is also supplied to the pre-abort handler. This is somewhat similar to interrupts from Alert-on-Update[37]. However those interrupts focus on monitoring the state of software TM metadata, and are not for general HTM.

However, a data conflict on a transactional but dirty line implies that the transaction has already done work that, if committed, could violate transactional semantics (unless there was false sharing, which is outside the scope of this work, or unless the conflict resolution policy is changed). Therefore, in this case we allow the HTM to automatically abort without pre-abort handler invocation, but we envision this behavior to be software-configurable to permit further uses of pre-abort handling, e.g. to avoid false-sharing-induced aborts.

### 3.2.2 Hardware Support

To support pre-abort handlers, the first hardware change that is needed is a mechanism to provide the processor core with the handler's address. One option is to reuse existing interrupt/exception vector tables and user-space signal handler mechanisms. Another way, which we use in our evaluation, is to add a new register [38], which we call the `Pre-abort Handler Entry` register. To put the address of the handler into this register, we can use an existing instruction for accessing control registers or, if no such instruction exists, introduce such an instruction.

The hardware already must track whether a thread is inside a transaction or not, e.g., to control the behavior of reads and writes. We add a third possible state, `HANDLER`. We change the hardware such that, when an abort-causing condition is detected by a thread in the `TRANSACTION` state, it doesn't automatically discard speculative state and transition to the `NON-TRANSACTIONAL` state. Instead, it saves the PC and transitions to the `HANDLER`

state.

As the system enters the HANDLER state, the stack pointer is also advanced by the cache line size to prevent the handler's non-speculative accesses from interfering with the transaction's speculative data. Without such precautions, the handler and the suspended transaction may end up sharing the same cache line (the top of the stack). This will result in the line being in some sort of anomalous state.

When entering the pre-abort handler, the PC is set to the entry point of the pre-abort handler (effectively jumping to that instruction). The arguments of the handler function need to be set as well, which we do so by following function calling convention and setting R1 to the abort cause and R2 to the abort-specific additional argument.

While in the HANDLER state, memory is accessed speculatively or non-speculatively depending on the status of the cache line. Reads or writes to cache lines with the transactional bit set are done speculatively, whereas non-transactional lines and lines not being tracked yet (e.g., cache misses) are done non-speculatively [39, 40]. In other words, the transactional bit is left as-is for each access [1].

The eventual fate of the handler can be one of three things. First, if the handler returns without the transaction getting committed or aborted, the transaction is resumed. The saved PC is restored and the stack pointer is moved back one cache line. If the abort was due to a clean data conflict, the line is added back to the read set. Lastly, the transaction state is changed from HANDLER to TRANSACTION.

Second, the handler may call the transaction commit instruction. In this case, the thread commits all speculative state as usual and moves from the HANDLER state to the NON-TRANSACTIONAL state. However, the thread has not yet left the handler function; returning triggers the same stack pointer and PC adjustment as in the "resume" case.

Lastly, the thread can encounter another abort-causing condition while still inside the pre-abort handler (*nested abort*). For example, while handling a system-call event a data

[1] This means that the handler can silently modify speculative state.

conflict occurs. In this case, we make no attempt to handle this nested abort situation and take the normal abort path (i.e. discard speculative content and call the regular abort handler).

In summary, while in the `HANDLER` state, the hardware behaves as if it's executing a transaction, *except*:

- The stack pointer is advanced and returned when entering and exiting the handler.
- Detecting additional abort causing conditions will abort the transaction.
- Accesses to cache lines not yet tracked as part of a transaction's read set or write set are done non-speculatively.

The `HANDLER` state is similar to transaction suspension provided by IBM's Power architecture [1]. However, in our scheme the suspension is done as part of the pre-abort handler, instead of being done explicitly.

It is important to note that the changes proposed above are isolated within the processor core itself, including both the TM hardware and private cache, and no changes are needed for the coherence protocol. In addition, if the entry point is not registered, there will be no handler call. The program behavior in this case is identical to the baseline: speculative state is discarded and control passed to the (regular) abort handler, allowing for backward compatibility.

In the following sections, we describe various use cases of the pre-abort handler.

## 3.3 Pre-abort Handler Use Cases

### 3.3.1 Handling System Call Aborts

Including system calls inside transactions is problematic because they are hard or impossible to undo [9]. As a result, conventional HTM systems [2, 3] simply abort when encountering system calls, forcing the transaction to discard the work done so far and take the

fallback path. The thread then re-executes the code before the system call, although now inside a critical section.

However, system calls are highly useful in debugging and for other purposes [41]. Indeed this is one reason why the C++ Transactional Memory proposal included the `optimize-for-synchronized` keyword to handle transactions that include system calls. Without them, a TM-aware compiler will fail to generate transactions and use critical sections instead [42].

By using pre-abort handlers, instead of automatically aborting, we can atomically *convert* the transaction into a critical section. Specifically, the handler is invoked when a transaction attempts a system call. This handler then attempts to acquire the fallback lock. Because all transactions hold the lock status in their read sets (❶ in Figure 3.1), the acquisition will force the other threads to abort any active transaction and prevent them from starting new ones. Once the lock is acquired, the thread then commits its work in isolation and returns to the system call instruction. As a result, the transaction's already-performed work can be considered to logically happen inside the critical section. At the end of the (now) critical section, the overall state of the thread is equivalent to what would have happened if the fallback path was used instead of starting the transaction.

This conversion of transactions into critical sections is similar to Irrevocable Transactions [43] that were used in the context of a lock-based software transactional memory (STM). However, instead of relying on the STM implementation's centralized contention managers and software locking protocols, pre-abort handlers provide similar benefits for best-effort HTMs.

Figure 3.3 shows the code, somewhat simplified for clarity, of how we might implement this idea using pre-abort handlers. When a transaction encounters a system call instruction, under the baseline system, the transaction is aborted and speculative state automatically rolled back. With pre-abort handlers, the instruction is instead quashed, just as is done for ordinary exceptions. The thread's control is then passed to the pre-abort handler.

```
1  handler(cause, argument) {
2    if(cause == SYSCALL) {
3      rc = trylock(fallback_lock);
4      if(rc == SUCCESS) {
5        HTM_COMMIT();
6        in_fallback = TRUE;
7        return;
8      } else {
9        // Lock taken by another thread
10       HTM_ABORT();
11 }}}
```

Figure 3.3: Pre-abort Handler For System Calls

The handler code first checks the reason for the exception (line 2) and then requests the fallback lock (line 3). When the lock is acquired and transaction successfully committed (line 4 – line 6), the transaction can no longer conflict with other transactions. The fallback lock is acquired and held by our thread, so active transactions are aborted and new ones prevented from starting. Since speculative state is also committed, the thread is now outside of a transaction, and the thread's state is as if it had been executing all along as a critical section.

When the thread returns from the handler, it continues executing non-speculatively, starting with the system call operation that originally triggered the conversion. The thread continues execution until the end of the "critical section," at which the fallback lock is released, just like it would be if the critical section had been executed using the fallback path in the first place.

Note that it is possible for the lock acquisition to fail, because the lock was already acquired by another thread (line 8). In this case, the pre-abort handler simply aborts, which leads to the speculative state being aborted and the regular abort handler being called.

It is even possible (but extremely rare) for a thread to successfully acquire the lock, but get aborted before the transaction was able to commit. For example, a conflicting access in another thread may occur just before the handler reaches the commit point. Since

transactions inside the pre-abort handler are still vulnerable to data conflicts, the transaction will simply be aborted, and control diverted to the regular abort handler. The thread still holds the lock (non-speculatively), so the regular abort handler simply needs to start the critical section body.

In all of these unsuccessful conversion cases, although the thread's speculative work is not salvaged, the overall resulting behavior is correct; in fact, it is the same behavior that occurs without transaction conversion.

If needed, we can also check the cause-specific argument, which according to Table 3.1 is the system call number, and take different action depending on which system call was the trigger. However, we choose not to do so in this implementation.

One advantage of our solution is that transaction conversion does not need to be done ahead of time. Other solutions [1, 43] require code modification that inserts transaction conversion just before the system call is attempted. However, as we shall see next, there are other types of non-conflict aborts that are more difficult to predict.

Another advantage of our solution is that when a transaction encounters a system call, the transaction does not need to discard work and retry. This is because in some cases system calls are issued due to a rare condition being triggered, perhaps an error condition. However, the condition may no longer be valid when the transaction was aborted and restarted as a critical section. By allowing the transaction to seamlessly convert into a critical section right then, the program can be in better shape to act on the situation.

### 3.3.2 Handling Capacity Aborts

When a core executing a transaction runs out of buffer space, the transaction is unable to proceed further without compromising its ability to abort and roll back. In this case, the cache-overflowing transaction is forced to abort before proceeding further. These are called *capacity aborts*. Capacity problems have been observed to be a major cause of aborts in commercial HTM systems [8], and an obstacle to broader use of HTM [44].

In addition, capacity aborts can be especially detrimental to performance. Existing best-effort HTM systems handle capacity aborts in the same way as system call aborts: the fallback path is taken. The fact that a transaction has accumulated enough read or write addresses to overflow buffer capacity means that it has probably done a lot of work. All that work is discarded and re-done on the fallback path, simply because of a limitation in the hardware. Even though the transaction has not conflicted with another thread and the already-done work is correct up to that point. If we instead have a mechanism to catch the capacity overflow before it occurs, we can take abort-avoidance action (e.g. convert the transaction into a critical section), we can avoid discarding the work.

However, unlike with system calls, which have a clear point in the code where it is clear that an abort-causing situation is going to take place, capacity issues are more difficult to predict. For system calls, this means that mitigating action can be inserted just before the system call through manual inspection [43], or even through some type of compiler analysis.

For capacity aborts, any memory access may be a possible abort site, depending on the size and other characteristics of the transactional state buffer (e.g. the private L1 cache), and the actual mix of addresses accessed by the transaction, etc. [40, 36]. This type of interaction is implementation dependent and hard to predict ahead of time – in many ways this problem is similar to the cache performance analysis problem, which is usually measured at runtime [45] rather than predicted statically.

This makes our handler-driven mechanism much more important. Instead of relying on a prediction mechanism that may or may not be correct, the pre-abort handler is only triggered when the buffer capacity overflow is actually imminent. Other than including a check for capacity abort, the pre-abort handler for transaction conversion just before a capacity abort is very similar to the one for system calls (Figure 3.3).

### 3.3.3 Inserting Non-transactional Work

Another use case of pre-abort handlers is to include non-transactional work within a transaction. One challenge with debugging hardware transactions is due to isolation provided by the hardware [41]. Even if there is a bug within the transaction, the state is discarded on abort, lost to the developer.

If there was a mechanism to log thread state non-transactionally, this state will still be available after the abort, making it easier to write correct transactions. This can be done by either adding special memory instructions [39] that execute non-speculatively, or instructions that suspend and resume speculative execution [1]. Unfortunately, these proposals require special hardware changes.

In comparison, the same pre-abort handler mechanism can also be used to provide such functionality. To do so, the developer manually triggers the pre-abort handler, by calling the XABORT or TABORT instructions. The argument passed to this abort instruction is passed as an additional argument to the handler (see Table 3.1). This can be used to pass information about the current transaction to the handler. Section 3.4.2 contains an example handler, which non-transactionally logs actions done by the active transaction, which can be inspected later on.

### 3.3.4 Pausing Transactions on Fallback-lock Activity

When a thread is using the fallback lock, the correct behavior is to disallow other transactions from executing, either by aborting or pausing them. This is because the critical section executes non-speculatively and does not rely on buffering to provide isolation. If other transactions are allowed to continue, this will result in undetected conflicts, which would violate transactional guarantees, as described in Section 3.1. However, other in-flight transactions have not (yet) conflicted with the critical section, so aborting them can be inefficient.

Instead, it is actually sufficient to *pause* the other transactions, rather than abort them immediately [4]. When the other transactions are notified of the fallback lock being acquired, they are paused, prevented from doing any additional work that can update the read or write sets. Only when the critical section is completed and the lock released are the transactions allowed to complete.

By using pre-abort handlers, we can implement transaction pausing. We use the fact that the pre-abort handler can be called when transactional but clean data is written to. Since fallback locks are only read (at the beginning of the transaction) but not written to, when there is lock activity, this will lead to the pre-abort handler being triggered. Transactions can then wait inside the handler, waiting for the lock to become free again. Once the lock is freed, the paused transactions are allowed to return from the handler, and resume operation.

```
1  handler(cause, argument) {
2    if(cause == ADDRESS &&
3          argument == fb_lock) {
4      while(test_lock(fb_lock) == BUSY)
5        DELAY();
6      return;
7    } else {
8      // Unrelated conflicting write, abort
9      HTM_ABORT();
10 }}
```

Figure 3.4: Pre-abort Handler for Fallback-lock Conflicts

Figure 3.4 depicts a handler that handles conflicts on the fallback lock. The transaction, as shown in Figure 3.1, starts by testing the fallback lock status and adding the address to the transaction read set. Before the transaction completes, the HTM hardware is notified of a write to a cache line that was part of its read set. The current operation is quashed and the pre-abort handler is triggered.

Since the handler is passed the address of the data conflict, we use this to see if it matches the address of our fallback lock (line 2–line 3). If the address does *not* match (line 8), this means the pre-abort handler was triggered by a write to another address. This is a gen-

uine data conflict, and the pre-abort handler aborts the transaction right away (line 10).

If the address *does* match, this means there was a change to the lock status (line 4). The handler loops until the busy lock is now free, effectively *pausing* the transaction (line 4 – line 5).

Once the lock becomes free, the loop ends and the handler returns (line 6). The PC is now back at the original operation and resumes normal execution. The transaction resumes, as if nothing has happened.

Pausing transactions, rather than aborting them, still results in correct execution. As noted earlier, a transaction paused within the pre-abort handler is still vulnerable to data conflicts. A paused transaction can have a data conflict with the critical section in one of two ways: with data that transactions had accessed *before* the lock acquisition, or *after* the lock release.

The first type of data conflict occurs when data the transactions had accessed is later accessed by the critical section. In this case, this is a standard type of data conflict; if the conflict was on dirty data, the transaction is automatically aborted, if it was on clean data, this is a nested abort and the transaction will (again) be aborted.

The second type of data "conflict" occurs when the data accessed by the critical section is later accessed by the transaction. In this case there is no atomicity violation because such execution is equivalent to serialized execution of these transactions in completion-order: first the critical section completes and releases the lock, afterwards, paused transactions eventually commit, *after* the critical section.

An important property we are taking advantage of is the fact that lock states are *reversible*; a lock that is busy will later be released and become free. Transactions are paused only while the fallback lock is busy, but when the fallback lock is freed, they are allowed to resume.

When the fallback lock shares a cache line with other data or locks, this can result in many more data conflicts being detected. Although this can hurt performance, this does

not mean the program is incorrect. Transactions will be unnecessarily aborted or have their pre-abort handlers triggered unnecessarily. The `test_lock` operation will return that the lock is free, and the transaction will be allowed to continue, albeit delayed somewhat.

One might be tempted to take this idea further and allow transactions to continue, blocking only when they try to commit. This is called lazy lock subscription, as opposed to early subscription which is what we use. However, lazy subscription can lead to problems [46]. Specifically, note that the critical section runs non-speculatively, updating values immediately. If the other transactions are not paused, they will be observing inconsistent state, breaking atomicity of the critical section. Since this can lead to further incorrect behavior, we avoid taking this route and simply pause the other transactions while the fallback lock is held.

### 3.3.5 Pausing Transactions on Critical Section Conversion

Interestingly, pausing transactions can be combined with converting transactions into critical sections. In Figure 3.3, when a transaction is converted into a critical section, the other active transactions are aborted immediately. This is done to keep the relationship between fallback lock activity and transactions identical: when the lock is acquired all active transactions are aborted.

Sometimes, multiple transactions can request being converted into a critical section at the same time. However, under the previous version of the pre-abort handler, only the first transaction succeeds; the others are aborted. Instead of doing so, we can try to pause them instead. When multiple transactions need to be converted into a critical section, we allow each of them to do so (one after the other), instead of needing multiple attempts before finally running as one.

We can do so through a simple change to the handler in Figure 3.3. Instead of trying and giving up after the first attempt, lock acquisition is attempted repeatedly. The successful thread will be converted into a critical section, while the other threads will be paused,

waiting for the lock to become free again (or be aborted while waiting).

### 3.3.6 Other Possibilities

In addition to the cases discussed so far, one can think of additional possibilities for pre-abort handlers. For example, one might use such pre-abort handling to avoid transactional aborts due to "housekeeping" activity, such as maintaining thread-local memory pools. When the memory pool is exhausted, accessing the global pool can lead to data conflicts. Instead, the transaction can be suspended while the thread-local free memory pool is replenished from the global pool. This is safe because even if the transaction is aborted, this expansion of the local free memory pool is safe (and usually beneficial) to keep rather than undo.

As another example, we can implement transaction early release with pre-abort handlers as well [26, 47]. Early release allows transactions to drop addresses from its read set, allowing the transaction to ignore conflicts on such addresses. We can do so by maintaining a per-transaction set of "dropped addresses." Later, when there is a read-write conflict interrupt and the addresses are confirmed to be part of the dropped addresses set, the conflict can be ignored, allowing the transaction to continue.

## 3.4 Evaluation Setup

We evaluate pre-abort handlers using SESC, a cycle-accurate architecture simulator [24], with an HTM model based on SuperTrans [23] modified to more closely match Intel's TSX. The processor cores are 4-way out-of-order, with private L1 and L2 caches. The L1 cache is 32KB, 4-way set-associative, with hit latency of 4 cycles. The L2 is 64KB per core with a hit latency of 18 cycles. The shared L3 cache is 1MB, with a hit latency of 34 cycles. The memory takes a constant 200 cycle latency.

Each L1 cache block is augmented with a bit indicating its transactional status. If a line with the bit set is to be evicted from the cache set, this is a capacity overflow. If it receives

a coherence request, this indicates a data conflict.

When a transaction encounters an abort-causing condition, the hardware checks if the pre-abort handler entry point is set. If it is, the program control is switched to the pre-abort handler. If not, the transaction is aborted immediately and program control switched to the regular abort handler. If the pre-abort handler decides to give up, the transaction is also aborted and program control switched to the regular abort handler. The abort penalty we used is a pipeline flush.

Three different configurations are studied. The *Baseline* configuration does not have the pre-abort handler specified and aborts on system calls, capacity limits, and fallback activity. *Hndl* uses a pre-abort handler to avoid system call aborts and capacity aborts by converting the transaction into a critical section (see Section 3.3.1 and Section 3.3.2). *Hndl+* additionally avoids aborting on fallback lock activity by pausing transactions (Section 3.3.4 and Section 3.3.5). In all three configurations, the regular abort handler uses random linear backoff, and takes the fallback path after 12 conflict aborts or 1 system call abort or capacity abort.

In Section 3.4.1, we first use microbenchmarks to demonstrate the utility of using pre-abort handlers. The first microbenchmark studies the impact of system call aborts and the benefits of handling them through pre-abort handlers. The second microbenchmark studies the impact and benefit of handling capacity aborts.

We then look at an application, vacation, from STAMP [26] and add non-transactional logging using pre-abort handlers, and quantify the overhead required.

Lastly, we use the STAMP benchmarks in section 3.4.3 to further quantify benefits on capacity aborts and fallback lock activity aborts; STAMP does not include system calls or non-transactional work within transactions. We exclude kmeans and ssca2, because their read/write set sizes are small and constant regardless of input, limiting the number of capacity aborts we can catch. In addition, they do not have many aborts in the first place and would not benefit from fallback activity mitigation techniques either. For measuring

performance, we used the ”Time” output from the program itself, which shows the wall clock time of the application's parallel region.

### 3.4.1 Microbenchmark Results

We first use microbenchmarks to show what benefits can be realized using our abort mitigation techniques. In Figure 3.5 we see the results from the first microbenchmark, which has transactions that may issue system calls. An AVL tree with objects, each 64-bytes large (1 cache line) and with a unique search key, is generated. A list of random search queries is generated as well, with a 50% probability of missing, and distributed to each thread. Each thread issues (read-only) search queries against the tree, and when the key is not found, an error message is logged to stdout. We compare the throughput of each of the three configurations.

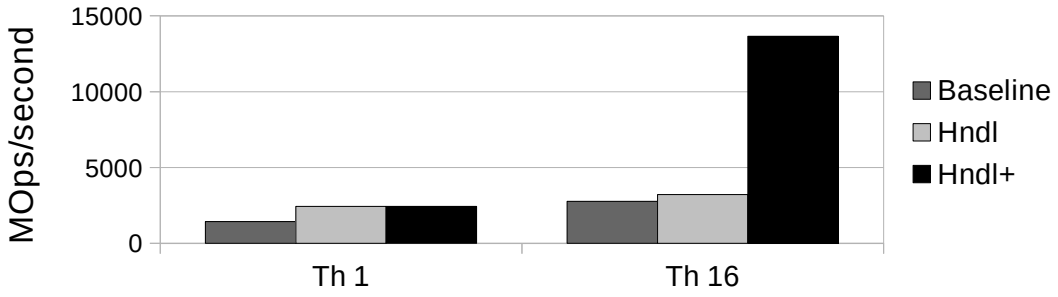
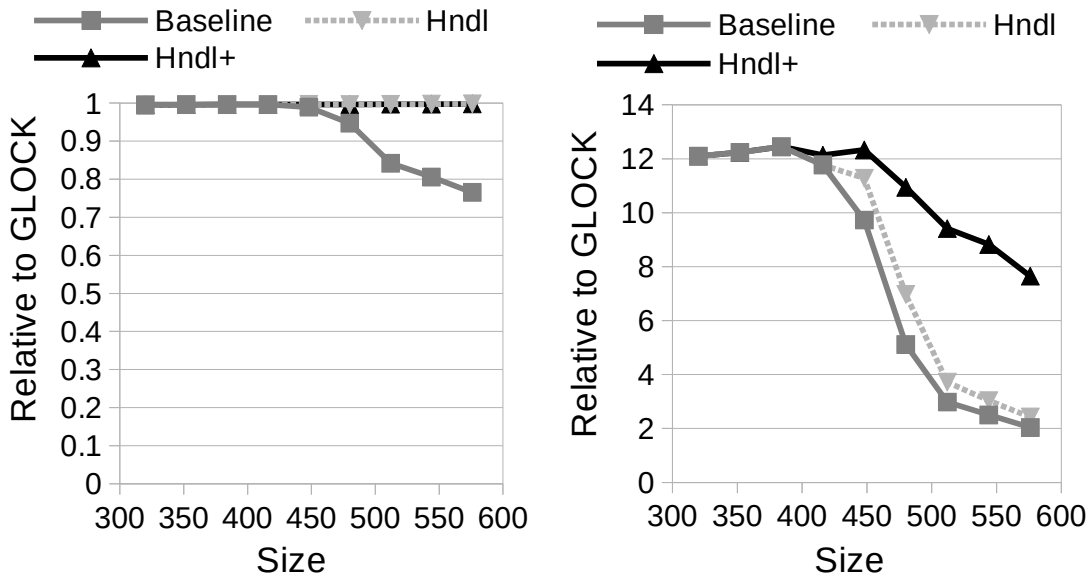



Figure 3.5: System Call Microbenchmark Results

From the results, we can see that there is a 70% speedup when using transaction conversion compared to the baseline when running single-threaded. This is because the microbenchmark issues a system call at the end of a transaction when it fails to find the query key. This means that a failed query will result in the thread executing a transaction almost to the end, abort due to the system call, and re-attempt the same code all over again.

At 16 threads, even though we have the threads all attempting transactions, under *Baseline* many of them eventually issue system calls and trigger further aborts on the fallback path. Unfortunately for *Hndl*, converting a transaction still results in the other transactions

getting aborted, even if none of them encountered any conflicts. By pausing these other transactions instead, *Hndl+* allows the microbenchmark to achieve a total of 9.5x speedup over the single-threaded result.




(a) 1 Threads

(b) 16 Threads

Figure 3.6: Capacity-scaled Microbenchmark Results

In Figure 3.6 we see the results from the second microbenchmark, which demonstrates capacity aborts. The microbenchmark has a randomly generated array along with a sequence of search queries generated beforehand. Similar to the previous microbenchmark, each entry is 64-bytes, and the search query has a 50% chance of missing. Each thread performs linear searches through the shared array, with one search per transaction. A larger search key will force the transaction to read more of the array and eventually run up against the capacity limit. We compare the throughput of the three configurations against *GLOCK*, which uses a single global lock instead of a transaction and therefore does not have any capacity issues.

When running with 1 thread, as we scale the input size, the performance of *Baseline* drops significantly, whereas the other two configurations are able to keep up with the global lock version. As we go from an input size of 448 to 576, *Baseline* suffers from a perfor-

mance dropoff of no performance loss to almost 24% performance loss due to increased capacity aborts. Using pre-abort handlers, we are able to salvage the work done by the overflowing transaction, and recover most of that performance loss, a 30% increase over *Baseline*.

When running with 16 threads, the impact of capacity overflow aborts becomes more prominent. In the same range of 448 to 576, *Baseline* goes from providing a 12x speedup over *GLOCK* down to 2x speedup, a 6-fold performance drop. If we allow transactions that hit the capacity limit to be converted into a critical section, the impact is reduced, resulting in a 10 – 30% performance improvement over *Baseline*. If we are able to pause them instead, as with *Hndl+*, we are able to greatly reduce the impact of the performance drop. The reason is because when one transaction hits the capacity limit, we are able to preserve the others by pausing instead.

In addition to improving performance for transaction sizes that are past the capacity limit, pre-abort handlers are able to provide a more graceful degradation just past that threshold.

### 3.4.2 Non-transactional Work Results

We investigate the benefits of including non-transactional work within transactions by modifying the vacation application from STAMP. Vacation simulates a travel reservation system, and randomly issues requests to an in-memory database: make reservation, delete customer, add and remove from item tables. We add a log entry to every customer deletion operation, as shown in Figure 3.7.

The manual abort is provided a `customerId` argument, which is then passed on to the pre-abort handler. The handler then logs this information, non-transactionally to standard output. Running with the simulator input, of the 4096 requests, 5% of the them resulted in a customer delete request. Logging these requests resulted in 2.0% runtime overhead, which is quite limited.

```
1  long bill = QRY_CUST_BILL(mngrPtr, customerId);
2  if (bill >= 0) {
3      HTM_ABORT(customerId);
4      DLTE_CUST(mngrPtr, customerId);
5  ...
6  void handler(int cause, int argument) {
7    if(cause == MANUAL) {
8      printf("[LOG] Customer %d Deleted\n",
9          argument);
10 }}
```

Figure 3.7: Adding non-transactional logging to vacation

### 3.4.3 STAMP Results

Figure 4.6 show the overall performance improvement provided by our mechanism when running STAMP, running with recommended simulator input. Since STAMP does not issue system calls within transactions, our experimental setup will only help on capacity aborts and fallback lock activity. We compare the execution times of the region of interest between *Baseline* and *Hndl+*.

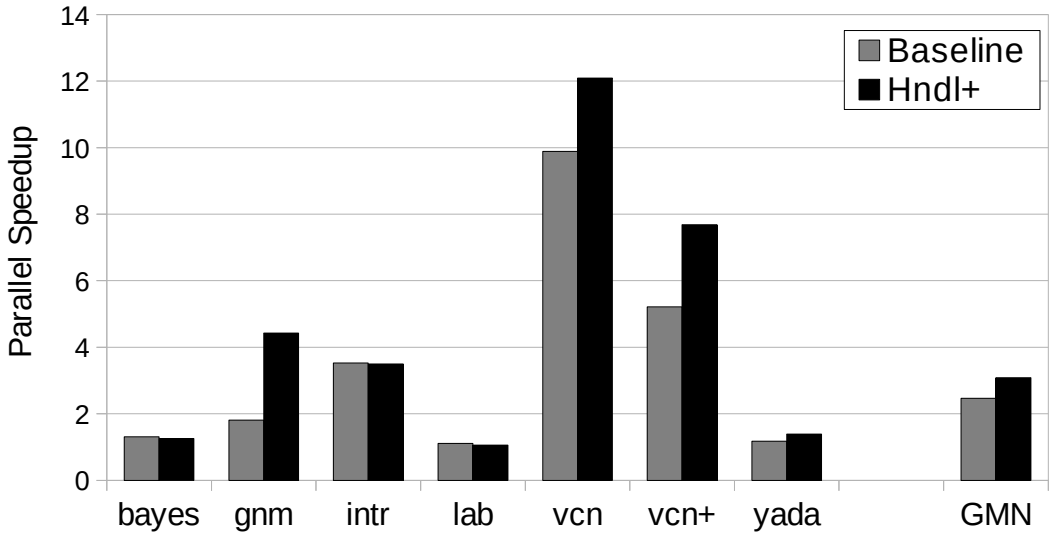



Figure 3.8: Overall Results

Using pre-abort handlers to convert transactions that overflow capacity, and pause them upon fallback lock activity, *Hndl+* shows an average speedup of 25% over *Baseline*. Genome

sees the most benefit (2.4x speedup). This is because a significant portion of its abort cycles are attributable to capacity-related issues, either directly because a transaction hits its capacity limit and aborts, or indirectly because capacity-aborting transactions later take the fallback path and abort other non-conflicting transactions. Vacation and yada also shows good gains (22%, 47%, and 18%).

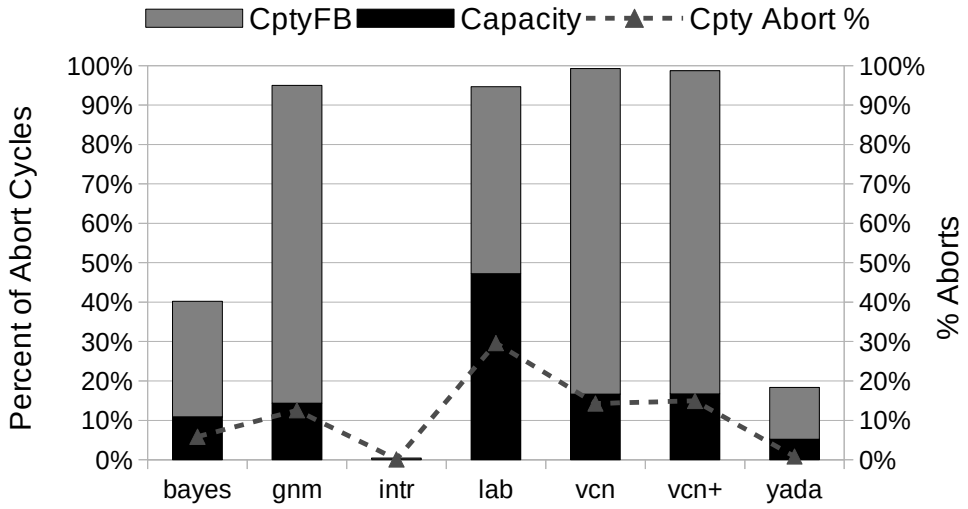


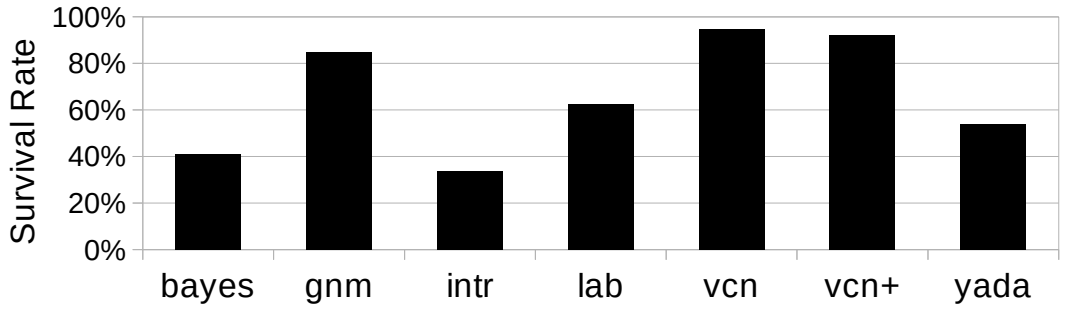

Figure 3.9: Contribution of Capacity Aborts

Figure 3.9 shows the percentage of all aborted cycles that can be attributed to capacity overflow when running the STAMP benchmarks with the baseline system. *Capacity* shows cycles directly attributable to capacity aborts, i.e. cycles spent reaching the capacity abort, as a percentage of all aborted-execution cycles (due to capacity, conflicts, etc.). *Cpty FB* shows the cycles indirectly attributable to capacity aborts: work lost in other transactions as they are aborted when the capacity-aborted transaction enters the fallback path. *Cpty Abort %*, shows the percentage of all transaction aborts that are directly attributable to capacity overflows (in other words, transaction attempts in this category contribute to *Capacity* cycles).

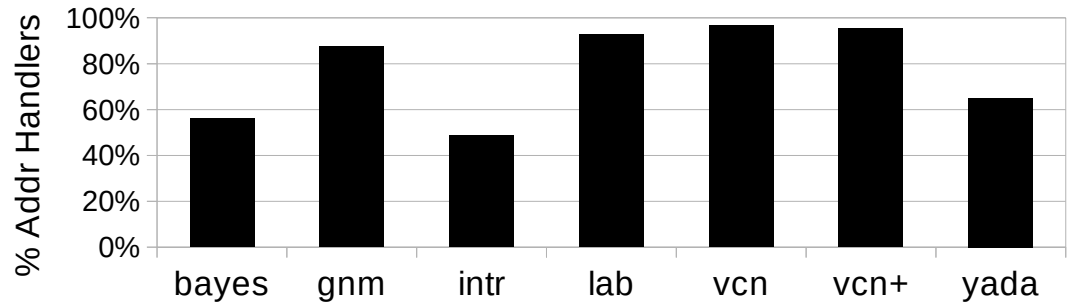
As we can see, in applications that do experience capacity overflow, the impact to lost work (cycles) is consistently larger than the fraction of capacity overflow aborts. This means capacity aborts are typically more costly than other aborts. Capacity aborts come

from a transaction having done a lot of work, even *too much* work, which is then discarded. In contrast, conflict aborts may happen very early in a transaction. The sole exception is labyrinth; capacity overflows occur so early in the transaction that threads are serialized and the benefit is minimal.

Interestingly, the results also show that that a significant fraction of aborted cycles are *in*directly attributable to capacity overflows. These aborts are caused by the fallback path being taken by overflowing transactions. As noted earlier, in the baseline system, transactions are required to abort when the fallback path is active, to ensure atomicity.




(a) Survival Rate




(b) Pre-abort Handlers Caused by Fallback Lock Activity

Figure 3.10: Other Statistics

Figure 3.10a shows the survival rate of paused transactions, i.e. the fraction of paused transactions that are eventually resumed (as opposed to being aborted while paused). A high survival rate indicates that the active thread (a critical section or a converted transaction) is less likely to conflict with paused transactions. These benchmarks can be expected to show increased benefit to those that have a lower survival rate. Indeed, genome and vacation have very high survival rates, and therefore see more benefit from pausing. Other

benchmarks have lower survival ratios and their benefit from pausing is lower.

Figure 3.10b shows what percentage of pre-abort handler activity was triggered by the fallback lock, compared to the total number of clean conflict handler calls (writes to transactional but clean data). Benchmark applications that have a high ratio means most pre-abort handlers were actually due to fallback lock activity. This is the case for many of our STAMP applications, although we can see that in bayes, intruder, and yada we see that many address handler invocations are due to actual data conflicts instead.

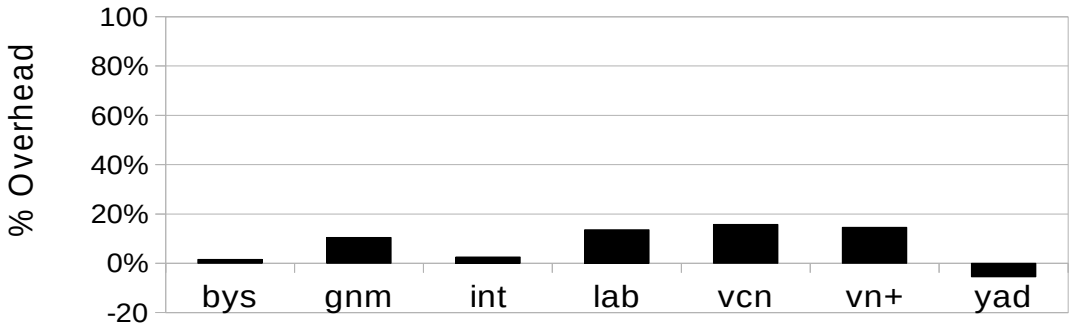



Figure 3.11: Overhead of Pre-abort Handlers

Because we call the pre-abort handler instead of automatically discarding a transaction's state and aborting it, one might be concerned about the overhead caused by the handler. To measure this overhead, we compared the runtime of the baseline implementation (automatically roll back the transaction) with an empty pre-abort handler (in other words, the pre-abort handler doesn't do any mitigating action and always aborts).

The results are shown in Figure 3.11, which shows that the overhead of invoking a pre-abort handler is relatively small and is easily outweighed by the performance benefits enabled by these handlers.

## 3.5 Related Work

Alert-on-Update [37] describes a mechanism that triggers an interrupt when a cache line is evicted from the private cache. It relies on new instructions that adds a cache line to an alert

set, which can cause an interrupt handler to be called when written to. Unlike our pre-abort handler, Alert-On-Update's handlers are limited to monitoring its software TM metadata.

Irrevocable Transactions [43, 48] is an STM system that converts a transaction into a special irrevocable mode. When a transaction is converted to this mode, it always has highest priority in data conflict resolution. At a high level this is similar to our transaction conversion. However, because it targets lock-based STM systems, the changes required are simpler. In addition, it does not need to support transaction capacity issues, which is difficult to predict without using pre-abort handlers.

Even though they are enclosed within a transaction, many memory operations are not required to be marked as speculative and be tracked by the HTM system. Some examples are read-only data or private stack data. Instead of treating all memory operations as speculative, some architectures such as ASF allow the software to selectively choose which operations are speculative, and which are non-speculative [39, 40].

Transaction Suspension [1] is a mechanism in Power's implementation that allows speculative execution to be temporarily suspended. Although this can also be used to take mitigating action against certain type of aborts, this requires the transaction to predict the abort situation and preemptively take action.

Between All and Nothing [49] proposes transactional resurrection, which allows aborted hardware transactions to resume. However, their default mechanism only steps in *after* the working set is cleaned up. The system then replays the software log and "resurrects" the aborted transaction. There is also mention of a "messy" version, which does not discard the working set (like with pre-abort handlers). However, it is only intended only intended for short delays, such as TLB misses, and is not discussed further.

System calls are challenging to support within transactions due to the side effects involved [50]. xCalls [9] allow transactions to use system calls, but require extensive software changes to work.

FasTM [13] tracks transaction metadata in hardware, but falls back to the software log

when a transaction exhausts its capacity. However, this requires the hardware understand how to manage the log and add an additional coherence state, which makes it less attractive for commercial HTM systems.

OneTM [35] introduces an *overflowed* state, which stalls other transactions while the first transaction attempts to complete. Conceptually, this approach is similar to paused transactions with transaction conversion, but requires transaction logging in software. TCC [4] is another approach related to transaction pausing. It allows other transactions to continue when a transaction encounters a capacity limit, blocking only their commit. However, commit blocking has problems similar to lazy subscription. In addition, TCC also requires a commit arbiter, which is not available in conventional HTM.

Proactive Transaction Scheduling [17] and Preventing versus Curing [19] model the probability of conflict and use this to guide a transaction scheduler. Similarly, Bloom Filter Guided Transaction Scheduling [18] use predicted read and write sets to avoid likely conflict aborts. These proposals focus mainly on predicting and avoiding data conflicts, and is largely orthogonal to our work.

Hybrid transactional memory [51, 52, 53] augments transactions running on conventional HTM with software TM. A transaction can decide before starting whether to run as software TM or hardware TM, and care needs to be taken to make sure data conflicts between transactions in either mode are properly tracked.

# CHAPTER 4
# FORGIVE-TM: SUPPORTING LAZY CONFLICT DETECTION IN EAGER HARDWARE TRANSACTIONAL MEMORY

## 4.1 Background

While a transaction is running, it needs to make sure it does not conflict with the speculative data of other transactions before it commits. Under requester-wins conventional HTMs, the data conflicts are detected through coherence messages [2, 3, 1].

When a transaction speculatively writes to a cache line, it sends a `GetM` coherence request message [54, 55]. Later, the same core may receive a coherence message destined for that line (i.e. `GetS` or `GetM`). This means that another thread is accessing the same data at the same time. This means transaction atomicity is broken, so the transaction is forced to abort.

Likewise, if a transaction has speculatively read from a cache line, it initially sends a `GetS` request. Later, when it receives a downgrade message (`GetM`), this means that another thread is writing to the same data. Here again the transaction is forced to abort.

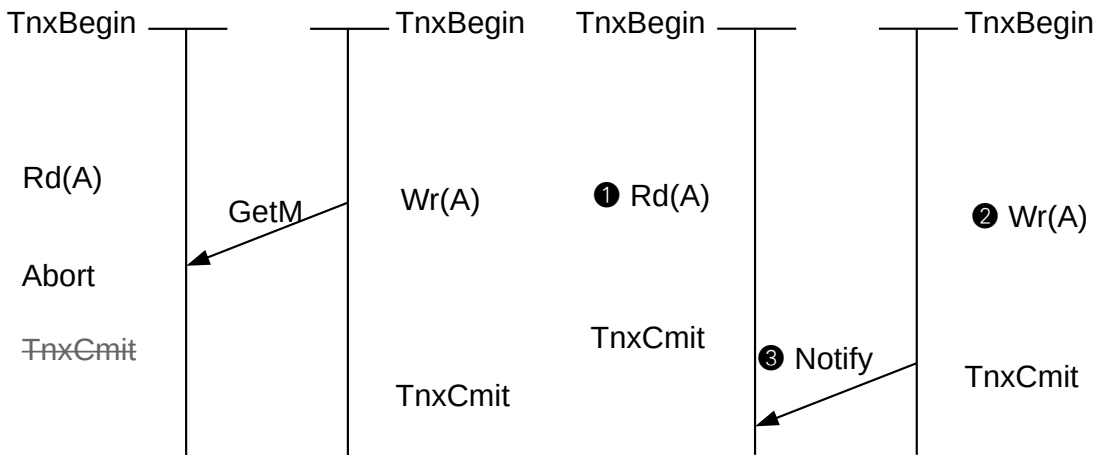
Figure 4.1a depicts what happens in detail. There are two transactions, and the lefthand transaction first issues a speculative read (①). Later, when the righthand transaction issues a speculative write (②), its core sends a `GetM` message, which is later received by the first core. This notifies the lefthand transaction that it conflicted with another core, which aborts itself (③).

As can be seen, the transaction is notified immediately of a data conflict. In other words, the conflict detection is done *eagerly*. The advantage of this mechanism is that it needs minimal support from the coherence protocol. From the protocol's point of view, the core has write or read permission, regardless of being done as a transaction or not.

In contrast, there are TM systems that do *lazy* conflict detection instead. Under lazy detection, transactions are not notified of data conflicts while executing. Data conflicts are checked for only at the end, when the transaction is ready to commit.

Figure 4.1b depicts how the behavior of lazy conflict detection TMs differ compared to that of eager conflict detection TMs. As before, we have two transactions, and each issues speculative reads and writes (❶, ❷). Note that at this point, neither of the two transactions notice that there is a data conflict.

Later, the righthand transaction is ready to commit. The transaction starts the commit phase and notifies the lefthand transaction which indicates a possible data conflict (❸). However, by this time, the lefthand transaction has already committed, so there is no abort. The lefthand transaction "happened" before the righthand transaction, and both committed without any issue.




(a) Eager Detection (b) Lazy Detection

Figure 4.1: Conflict Detection

Broadly, there are two mechanisms for implementing lazy conflict detection. First is the TCC family of HTMs [56]. These HTMs buffer all writes locally without telling the other cores. When the transaction needs to commit, commit permission is requested. Once this has been granted, the write set is broadcast. Other transactions compare their read set with this and get aborted if there is a conflict. The differences between implementations lie

in how commit permission arbitration is done and how the write set is broadcast.

The second is the EazyHTM family [57]. These modify to the coherence protocol to include message types which are used to construct local conflict sets. In other words, the conflict may be detected eagerly, but are not immediately acted on. On commit, conflicting transactions are sent explicit abort messages. More advanced implementations attempt to reduce the number of such messages.

As can be seen, both types of proposals require changes to the coherence protocol. The TCC solution requires commit arbitration messages, and the EazyHTM solution requires conflict messages. However, coherence protocols are difficult to design correctly, and every additional message type has the potential to significantly increase verification complexity [58].

Instead, we propose Forgive-TM, which allows lazy conflict detection, while keeping the existing coherence layer intact. The hardware changes required are isolated within the processor cores themselves, which can be independently tested and verified.

## 4.2 Related Work

TCC [4] are one of the first proposals for lazy transactions. It works by issuing writes locally without notifying the other cores. Later, when the transaction is complete, the core sends a commit request on the shared bus. The commit phase works by broadcasting the transaction's write set on the bus, aborting any conflicting transactions in the process.

Following TCC, there have been various extensions with the goal of improving the performance of the commit phase. Scalable TCC [59] and BulkSC [60] take a similar approach to TCC, but overcome the requirement of a shared bus by adding a commit arbiter. ScalableBulk [61] and BulkCommit [62] divide the commit phase into smaller pieces, called *chunks* so that multiple commits can be done in parallel.

Instead of adding an explicit commit phase, EazyHTM [57] adds explicit transaction abort request messages to the coherence protocol instead. While executing a transaction,

the core sends `txMark` requests and `txAccess` requests to notify other cores that its transaction is currently working on a given address. These messages are received by other cores, and allows the transactions to build a list of conflicting transactions. When the transaction is ready to commit, `abort` requests are sent to each conflicting transaction.

PI-TM [63] extends EazyHTM by adding bits called $pi$ bits to the private cache. EazyHTM has a disadvantage when recovering from a transaction abort: which part of the speculative data had actually conflicted with another core is not known, therefore abort recovery can be complicated. $pi$ bits can accelerate this phase; however, fundamentally PI-TM is similar to EazyHTM and has similar characteristics.

LV* [64] also tracks sets of conflicting transactions at each core. During execution, transactions snoop coherence messages to keep track of which other transaction it conflicts (a *killer map*). At commit, a commit message is broadcast to abort conflicting transactions, whereas at abort, an abort message is broadcast to its clear killer map entry in the other cores.

SI-TM [65] is a more ambitious proposal that provides *snapshot isolation* in hardware TM, instead of the usual *opacity* guarantees that other systems provide [55]. Snapshot isolation is a weaker correctness guarantees and therefore can allow additional scalability. In addition to snapshot isolation, SI-TM also provides lazy conflict detection, through multiversioned memory and transaction timestamps. However, supporting such a system requires extensive changes, including changes to the memory controller.

There have also been proposals that allow a mix of both eager and lazy transactions.

FlexTM [66] detects conflicts eagerly and keeps track of which transactions conflict with which in hardware. However, although *detection* is done eagerly, *management* can be done lazily. A software routine, instead of a hardware one, is called to determine what to do with the conflict, including explicitly aborting the other transactions at commit time.

DynTM [67] takes a similar approach to FlexTM. Lazy transactions detect conflicts eagerly, but the conflicts are managed lazily by sending explicit `AbortTx` requests at

commit time. Eager transactions on the other hand manage conflict eagerly, by aborting or stalling the conflicting transaction. Which mode to run as is determined in the hardware.

Speculation-Based Conflict Resolution [68] also detects conflicts eagerly but handles them lazily. Unlike other schemes, however, transactions with write-after-read conflicts are ordered. Later, when conflicting transactions attempt to commit, the commit is done in the write order, and aborts are avoided.

SEL-TM [69] is most similar to our work. Like Forgive-TM, SEL-TM allows for a mix of eager and lazy conflict detection and manages lazy requests through a lazy set (called a Lazy Address List) and a lazy store buffer. However, unlike our scheme SEL-TM only after the commit arbitration is done. In addition, in SEL-TM, once an access is done lazily it cannot be converted into an eager access. In addition, the eager access in SEL-TM is based off of LogTM, which allows for NACKing of conflicting memory access and therefore is less sensitive to data conflict aborts.

In general, prior work in supporting lazy conflict detection in hardware TM requires changes to the coherence protocol. However, coherence protocols are generally difficult to design correctly [58]. these designs design has a disadvantage that requires adding additional coherence messages and states, which complicates design [70]. In addition, the additional messages and states are specific to hardware transactions (i.e. not used outside of transactions), which acts as a disincentive for additional effort in validating design.

## 4.3 Forgive-TM

In comparison to the prior work on lazy conflict detection, Forgive-TM is a mechanism that *allows* support for lazily detected writes to be done by transactions, *without* modifications to the coherence protocol. The changes are limited to the existing TM hardware, which lies within the core itself, whereas the coherence hardware is kept the same. This allows hardware vendors to take a more gradual approach in providing hardware TM support.

What Forgive-TM does is "act first, ask forgiveness later [1]." Speculative writes are done with only read permissions, *not* write permissions. Only later, when the transaction is about to commit, are the write permissions acquired. This is allowed because as long as the operations collectively "appear" to happen instantaneously, the operations can be rearranged [55]. In addition, transactions that read speculative data are notified of any other writers, and can be aborted.

In detail, Forgive-TM divides speculative writes into two categories: writes done eagerly and writes done lazily. If the write is to be done eagerly, it will be done in the conventional manner. The processor first checks for write permission and acquires it if needed and updates the local private cache. Because acquiring write permission entails sending a coherence message (`GetM` messages), this notifies the other cores of any potential conflict eagerly.

Actually, most of the writes will be done eagerly.This is because not all writes need to be handled in this special manner [69]. Many of the writes will be to private data. However, some of the writes will be to data that many of the threads will needed to access. In addition, each additional write done lazily can impact the commit latency can add up [71].

If not eagerly, the write is to be done lazily. Under this scheme, the processor initially requires read permission only (by sending `GetS` requests). This will allow the transaction to be notified of any other writers and abort. However, it unlike in the conventional case, it does *not* require write permission, but instead keeps silent of its own (lazy) write.

The address of this cache line is then added to a set, the LazySet. Later, before the transaction commits, the processor requests write permission (by `GetM` requests). In other words, the lazy write is now *converted* to an eager one. Any other transactions that also accessed the line will be aborted. From the core's point of view, although the operations are rearranged (the write permission request was delayed), the end result is still the same: the core had speculatively updated data with the correct write permissions.

[1] also known as "it's easier to ask forgiveness than it is to get permission" – Grace Hopper

Note that when the conversion to eager write is not specified. It can be done at any time, as long as the transaction is not completed ("committed"). This means that a write can be done lazily at first, but later released as an eager write to make room for another lazy write (the address was *evicted* from the LazySet).

At commit time, the hardware begins the commit operation. Commits are divided into two phases. The first phase is the *Commit-Prep* phase. During this phase, the hardware steps through each entry of the LazySet and acquires write permission for each of them (by sending the `GetM` request).

From the recipient's point of view, this is as if data conflict detection was done later, or "lazily." Recall that transaction data conflicts can occur between a read and a write operation, or two write operations. Since write operations are not announced to other cores until later, the conflict is not detected until the `GetM` messages are sent.

As noted earlier, not all writes are equal. Some do not need to be done lazily, while others will see benefit. Forgive-TM includes a scoring mechanism that determines which writes to be done eagerly and which to be done lazily (details explained later in Section 4.3.3).

### 4.3.1 Hardware Overview

Figure 4.2 shows the hardware overview of our proposal. The items in white are common with the baseline commercial HTMs. Each processor has HTM hardware that manages the transaction state and checks for any data conflicts. The private caches contain the `Data` and `Dirty` bits for each cache line, and additionally have `Tnx` bit, which indicates whether the cache line was accessed speculatively or not. A speculatively read line has only the `Tnx` bit set, whereas the speculatively written line has both `Tnx` bit and `Dirty` bits set.

The items in gray are new. First is the `LazySet`, which contains a set of addresses that were written to lazily. At the start of the transaction, this set is empty. As the transaction issues more and more writes, the set fills up. Each time a new entry is added to the

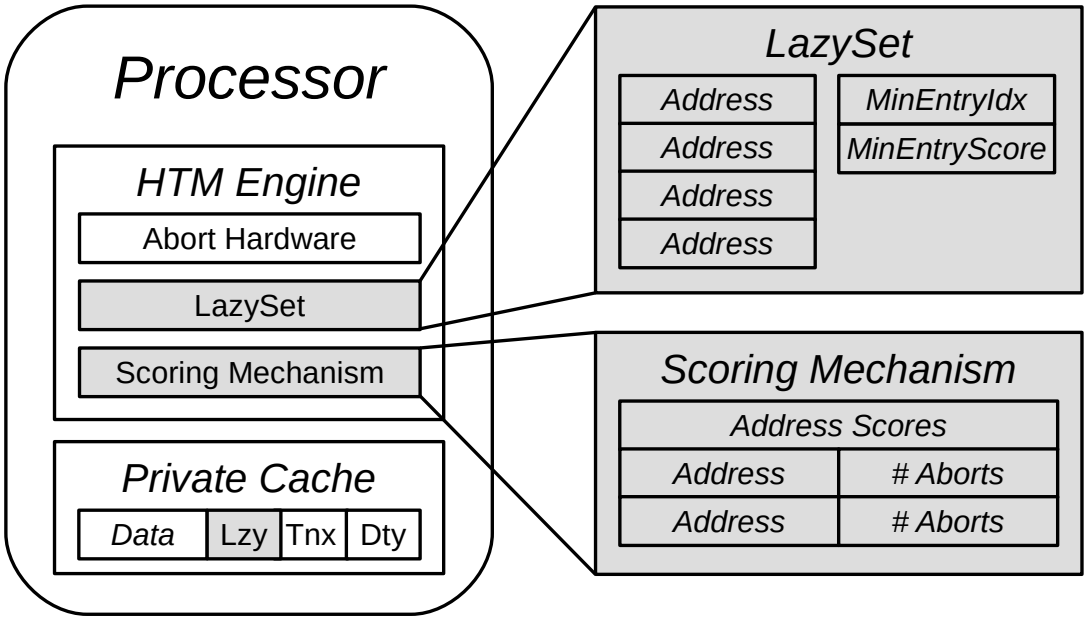



Figure 4.2: Architecture

set, its score is computed using the `Scoring Mechanism` and the `MinEntryIdx` and `MinEntryScore` is updated. Later, when the transaction is complete the *Commit-Prep* phase pulls entries from the `LazySet` one by one and issues `GetM` requests.

Second, is the `Scoring Mechanism`. Because the `LazySet` is of limited size, each write operation is computed a score. The score is a proxy that attempts to model the probability that the destination address will be involved in a data conflict with another core. The higher the score, the more likely it is to experience a data conflict.

The scores are computed using various statistics collected over the course of the program. In Forgive-TM, we use the number of aborts caused by a data address as the overall score of the operation. However, other variables can be used as well, such as the PC of an write instruction that is likely to cause aborts, and so on. How exactly the scoring is done and the table populated is discussed in more detail later in the paper.

Each cache line is also augmented with a `Lazy` bit. This bit indicates whether the line has been written to lazily or not.

### 4.3.2 LazySet Maintenance

The `LazySet` is a structure that stores a set of addresses (of cache lines) which were lazily written to. This set serves two purposes. First, it determines whether an address was previously written to. Second, it determines which addresses the *Commit-Prep* phase needs to handle.

During transaction execution, every time a write operation is about to be done, the `LazySet` is consulted to see if the address was already added. If so, this means that the address was already written to lazily before, and the cache line is already part of the speculative store (the private L1 cache in our case). All subsequent writes to this cache line is to be done lazily.

If the address is not already part of the `LazySet` and misses in the private cache, this is a new write operation. As long as the `LazySet` has space available, all writes are initially done as lazy. The destination address of the operation is added to the set, and the score of the operation is then computed using the `Scoring Mechanism`, and updates the `MinEntryIdx` and `MinEntryScore` if needed.

when the *LazySet* is full and misses in the cache will need to see if its eligible for lazy writing. The score of the new write operation is computed, and if it is higher than the smallest entry already in the `LazySet`, the old entry is *evicted* from the set, and destination address of the new write takes its place. The old entry is *converted* into an eager write, by sending `GetM` request for the address. The new entry is done lazily as usual, by acquiring read permission instead of write, and getting added to the `LazySet`.

On the other hand, if the score of the new write is smaller (or equal to) the smallest entry, then the new write operation is the one that's done eagerly.

Although both the private cache and the LazySet can experience evictions, the end result is different. If a cache line was evicted from the private cache and the line was speculative and dirty, the transaction needs to be aborted. This is because if the line is evicted, there is no way to recover the speculative data. This is the case for both writes done eagerly or

lazily. In addition, because the transaction is aborted, this means all lazily written lines will be always exist in the private cache.

### 4.3.3 Scoring Mechanism

Not all writes behave in the same way [69]. Some of the memory writes are for managing the call stack, for pushing and popping function local variables. Other writes will be to private data. However, some of the writes will be to data that many threads will share, and it is these writes which transactional memory should be interested in.

In addition, specifically for HTMs that detect and handle conflicts lazily, the number of writes can directly impact the commit latency. Each lazy write results in more operations that need to be handled during the commit phase [71, 69].

As a result, Forgive-TM limits the size of the lazy set and includes a mechanism to rank write operations and do lazy writes on only a subset of them. Each write operation is given a score. The score acts as a proxy to the likelihood that the write operation will trigger a data conflict abort. The higher the score, the more likely doing the write eagerly will result in data conflicts (lazy writes can result in data conflicts as well, but as shown in Figure 4.1b, writes done lazily can reduce many of them).

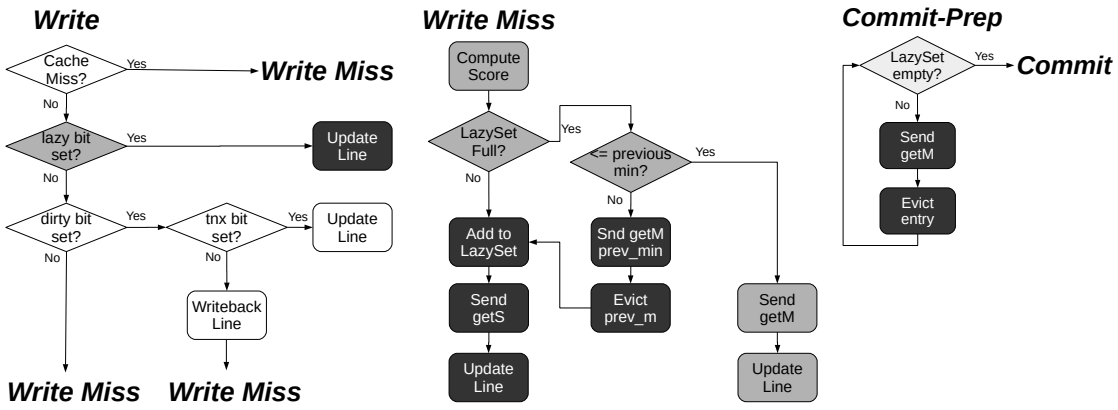
Forgive-TM use the number of aborts triggered by the given destination address as the write operation's score. The number of aborts are obtained using the following mechanism. For each incoming coherence request that results in a data conflict abort, the corresponding entry within the table is incremented. Since there can be an unlimited number of addresses that can result in such an abort, entries can be evicted from the address table, least-recently-used entries first.

For each new write operation, the score is computed and compared with existing entries already in the `LazySet`. If the lazy set has space available, we always do the write lazily. If, on the other hand, the lazy set is already full, the newly computed score is compared against the minimum score of the set. If the new score is greater than the current smallest

entry, then the entry is evicted and replaced with the new one. On the other hand, if the new score is equal to or smaller than the current smallest entry, then we do not consider the operation for lazy write, and do it eagerly instead. Note ties result in favoring the older entry, since keeping the older entry will allow for larger overall laziness for the entire transaction.

There are other possibilities in computing the score of these operations, and in Section 4.4, we study other alternatives.

### 4.3.4 Operation Flowcharts




(a) Writes (b) Write Misses (c) Commit-Prep

Figure 4.3: Operation Flowcharts

In this section, we describe in detail how certain operations are handled. First, Figure 4.3a depicts a flowchart of how write handled under Forgive-TM. The items in white are common with conventional HTMs, whereas the ones in gray are new to Forgive-TM. Dark gray items are specific to the lazy write path, whereas light gray items can be for either eager writes or lazy writes.

When a transaction executes a speculative store, the processor first checks if the cache line already exists in the cache. If the line does not exist, the processor starts the write miss procedure, which is shown in more detail in Figure 4.3b.

If the line does exist, the processor then checks the lazy bit. If the bit is set, this means

the line was previously written to lazily. In this case, the current store operation simply updates the cache line data (lazily) and completes.

If the bit is *not* set, the dirty bit is next checked. If the bit is not set, then although the cache line does exist in the cache, the coherence permissions might indicate that we have write permission. Therefore, the write miss procedure is taken.

On the other hand, if the bit is set, then we further check if the transactional bit is set. If the bit is clear, this indicates the contents of the line was written to prior to the transaction. In other words, the data contents of this line is *not* speculative. Therefore, the data needs to be written back before the write miss procedure is handled.

If the transactional bit is also set, then we simply update the line. This case indicate a line that was eagerly written, being written to again.

Figure 4.3b specifically details a flowchart of how write misses are handled are done under Forgive-TM. When the processor needs to acquire the cache line and/or coherence permissions for speculative writes, it goes through the process shown here. First, the `LazySet` is checked to see if space is available. If the `LazySet` is not full, the write can be done lazily. The `ScoringMechanism` computes the score for this write, and add the address and the score to the `LazySet`. The processor then sends a request for read permission, a `GetS` request, if needed, and updates the data contents of the line.

On the other hand, if the `LazySet` *is* full, then we might need to evict an existing entry. The newly computed score of this write operation is compared against the current minimum entry (`prev_min`) within the lazy set. If the score is greater than the previous min, then this operation has higher probability of being a source of data conflicts than the previous min. The previous min is *converted* into an eager write, by requesting `getM` permissions. Once this is complete, `prev_min` is evicted from the `LazySet` and the new address is added.

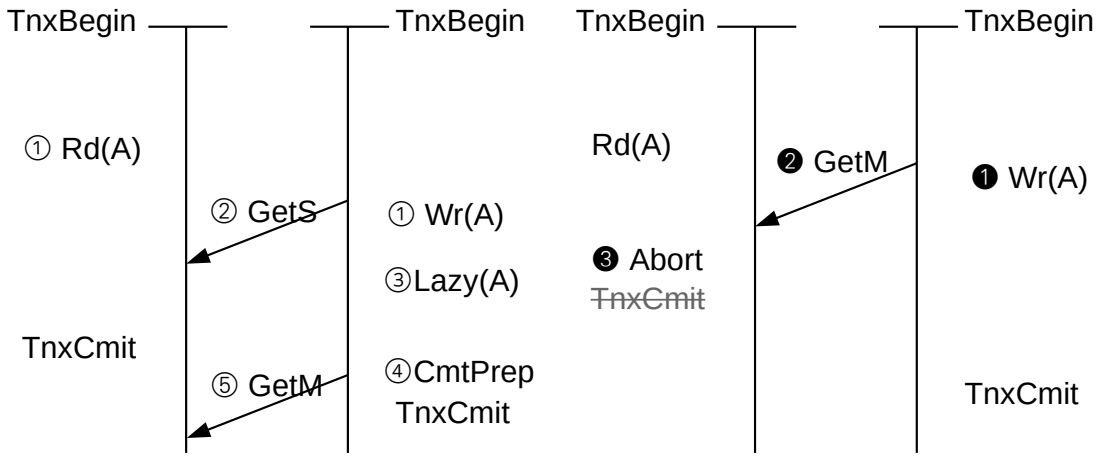
If the new score is *not* greater than the score of `prev_min`, this means the new write operation has low probability of triggering a conflict abort. The write operation is done as

usual, by sending the `getM` request and updating the cache line contents.

Figure 4.3c looks at what needs to be done in the `Commit-Pref` phase. Recall that at the end of a transaction, Forgive-TM inserts an additional phase, the `Commit-Prep` phase to ensure that all lazily written lines are accounted for. Each entry within the lazy set requests write permission, one after the other. All of this is while still inside the transaction, but before the official commit phase. Once all of the addresses are properly converted into eager writes, the regular commit starts.

### 4.3.5 Examples

In this section, we go through a few examples and show how various operations work under our proposal.




(a) Writing Lazily (b) Ineligible Lazy Write

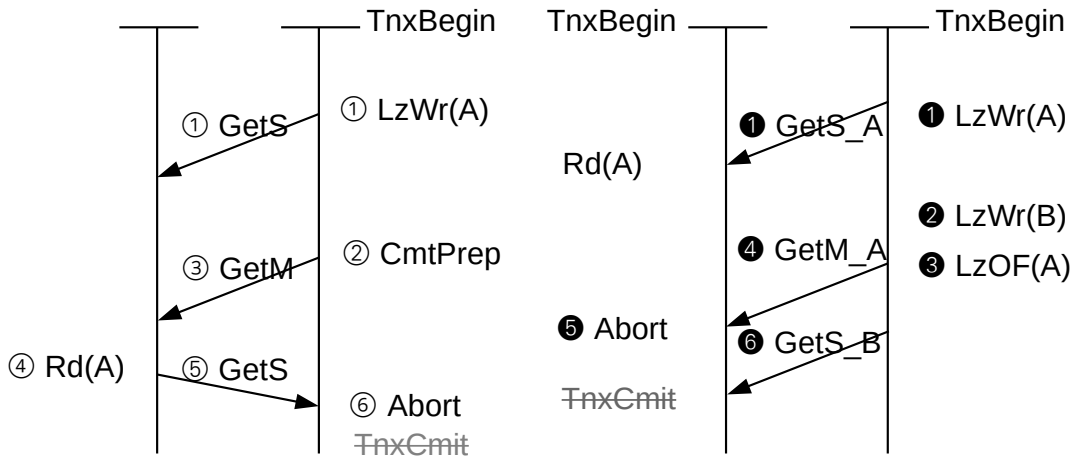
Figure 4.4: Timeline of Writes in Forgive-TM

First, Figure 4.4a depicts an example of how writes can be done lazily. Like before, we have two transactions that each do speculative read and writes (①). This time however, the core issues a `GetS` request (②) instead of a `GetM` request. In other words, the core is requesting read permission only, not write permission (yet). At the same time, the address `A` is added to an internal set, called the lazy set ③.

Later, when the transaction is ready to commit, the TM hardware starts the Commit-Prep phase (④). The TM hardware steps through each entry within the lazy set and issues `GetM` requests to each (④).

Once each line is properly accounted for, the TM hardware starts the regular commit operations. Since from the core's point of view, the transaction is complete, with read and write permissions for each core. In other words, the transaction's state is as if all write operations were done eagerly, and there were no conflicts. Therefore, the commit operation remains unchanged.

On the other hand, sometimes lazy writes can be deemed ineligible. This case is depicted in Figure 4.5b. The first transaction is attempting to do a lazy write (❶). However, the HTM hardware determines that the address is *not* eligible to write lazily. This can be because the address had a low score in the scoring table, for example. In this case, the core issues a `GetM` request instead of a `GetS` request, since the write is done eagerly (❷). This request is received by the other transaction, which subsequently aborts (❸).




(a) Transaction Abort (b) Lazy Overflow

Figure 4.5: Other Scenarios

Figure 4.5 depicts various cases that can happen under Forgive-TM that can differ from baseline.

The first case depicted in Figure 4.5a is a transaction failing to commit. Initially, the

transaction proceeds as normal, by doing a lazy write and sending a `GetS` request (①). Later, the transaction starts the Commit-Prep phase (②). This steps has the HTM hardware step through each entry in the lazy set and requesting write permission for each line (③). However before all of the addresses in the lazy set were successful in acquiring write permission, let's say another core requested read permission for the same line (④). This will cause the other core to send a `GetS` request (⑤). The first transaction, before it was able to commit, receives this coherence message and aborts as a result (⑥).

Under baseline, speculatively written lines can abort transactions when they are evicted from the private cache. This is because when the line is evicted, there is no place to store the speculative updates. Under Forgive-TM, this does not change. Any lines written speculatively, either eagerly or lazily, will abort transactions when evicted. This means that if a address is added to the lazy set, the line will always exist in the private cache.

The second case in Figure 4.5b shows a case where the lazy set had overflowed. In ❶, the transaction issues a lazy write to address A, which leads to a `GetS` request being sent for that address. Later, the transaction tries to issue a write to address B ❷. Unfortunately, the lazy set is already full at this point. The scoring table indicates that the new write to B is of higher priority than the older write to A. Therefore, the address A is evicted from the lazy set (❸, ❹). This results in the `GetM` request being sent for address A, which can lead to the lefthand transaction detecting the conflict eagerly, and aborting (❺). The transaction then reattempts the write to B, which is done properly as a lazy write with a `GetS` (❻).

### 4.3.6 Discussion

Table 4.1 shows an overview of the various types of conflicts that can occur under Forgive-TM and how they are handled. Each row indicates the state of the operation that the transaction is currently attempting. It can be one of four cases: eagerly loading (`E-Load`), eagerly storing (`E-Store`), lazily storing (`L-Store`), or attempting to commit a lazily written address (`L-Cmmt`). Each column on the other hand indicates the state of the ad-

dress as viewed by the other transaction. It can be either eagerly read (`E-Read`), eagerly written (`E-Wrtn`), or lazily written (`L-Wrtn`).

Table 4.1: Types of Data Conflicts

| | **E-Read** | **E-Wrtn** | **L-Wrtn** |
|---|---|---|---|
| **E-Load** | N/A | Abort | Delayed |
| **E-Store** | Abort | Abort | Abort |
| **L-Store** | Delayed | Abort | Delayed |
| **L-Cmmt** | Abort | Abort | Abort |

When a transaction is attempting to eagerly load an address, it can encounter 3 different state: the address was eagerly read, eagerly written, or lazily written. If the address was eagerly read, there is no data conflict to worry about. If the address was eagerly written to, the transaction triggers a data conflict abort, right then and now. However, if the address was lazily written to, the data conflict is delayed. The address looks as if it was read from, not written to, and therefore no conflicts were detected. Only later, when the other transaction attempts to commit is the data conflict exposed.

When a transaction is attempting to eagerly write to an address, the other transaction is always aborted, no matter which state the address was in.

When a transaction is doing a lazy write, is appears as an eager read to the other transaction. Therefore, transactions that did an eager write to that same address will be aborted. However, those that only read from the address, or did an lazy write, will not notice the conflict until later.

Last, when the transaction is finally at the end, the `Commit-Prep` phase is started and all lazy writes need to be converted into eager writes. The core acquires write permission for all of them, and aborts any concurrent transactions that accessed the same address.

In summary, under conventional eager HTMs, any pair of operations that are not load operations can lead to a data conflict abort. However, under Forgive-TM, several of these operations can delay the detection of the data conflict. This can allow much better parallelism, because the window of data conflict is now tightened to during the `Commit-Prep`

phase, and futile aborts, transactions that are aborted by transactions which are themselves aborted, can be avoided [71].

### 4.4 Evaluation and Results

To evaluate the effectiveness of detecting writes lazily, we modified an architecture simulator to support hardware transactions similar to Intel's TSX [24, 23]. The simulated architecture is a 8 core, 4-way out-of-order processor with private L1 and L2 caches. The L1 cache acts as a private cache for speculative data, while the l2 cache stores the original data. The L3 cache is shared with all cores.

For applications, we use the STAMP benchmarks, with updates to more closely match the draft C++ transactional memory support [26, 72]. Heap allocations calls were converted to use per-thread pools to reduce data conflicts on these items. Transactions were also augmented to support randomized linear backoff, and after 12 conflict aborts the fallback path is take. All experiments were run with recommended simulator input.

Table 4.2: Benchmark Characteristics
* indicates average per transaction

| | **NumTnx** | **ReadSet*** | | **WriteSet*** | | **HotAddrs*** | | **TopAddr(%)** |
|---|---|---|---|---|---|---|---|---|
| | | AVG | MAX | AVG | MAX | AVG | MAX | |
| **yada** | 3521 | 38.4 | 366 | 15.0 | 168 | 4.4 | 194 | 14 |
| **bayes** | 621 | 87.2 | 795 | 42.9 | 461 | 5.6 | 202 | 44 |
| **intruder** | 11267 | 17.6 | 53 | 4.4 | 24 | 1.6 | 35 | 96 |
| **labyrinth** | 229 | 17.7 | 50 | 6.7 | 37 | 0.4 | 1 | 100 |
| **genome** | 5914 | 34.6 | 100 | 5.1 | 30 | 0.4 | 47 | 29 |
| **ssca2** | 47277 | 10.1 | 11 | 3.9 | 4 | 0.07 | 1 | 55 |
| **kmeans+** | 8214 | 12.6 | 15 | 2.97 | 3 | 6.2 | 8 | 90 |
| **kmeans** | 10952 | 12.6 | 15 | 2.97 | 3 | 8.4 | 14 | 96 |
| **vacation+** | 4096 | 81.0 | 129 | 10.8 | 19 | 0.06 | 10 | 57 |
| **vacation** | 4096 | 68.6 | 92 | 10.3 | 14 | 0.02 | 10 | 47 |

Table 4.2 describes some characteristics of the benchmarks. *NumTnx* indicates the number of transaction instances in the benchmark (with the simulator input). *ReadSet*

and *WriteSet* show the size of the read set and write set. We show both the arithmetic mean and the maximum set size for both of these. *HotAddrs* show the number of times a transaction wrote to a "hot address." By hot address we mean an address that has ever been responsible for a conflict during the entire execution time of an application. Last, we show *TopAddr*, which looks at how skewed conflict abort addresses are. This is computed by first generating a histogram of how many aborts each address was responsible for. Once the top 10 addresses are generated, the total number of aborts these addresses are responsible for is computed. This total is divided by the total number of conflict aborts.

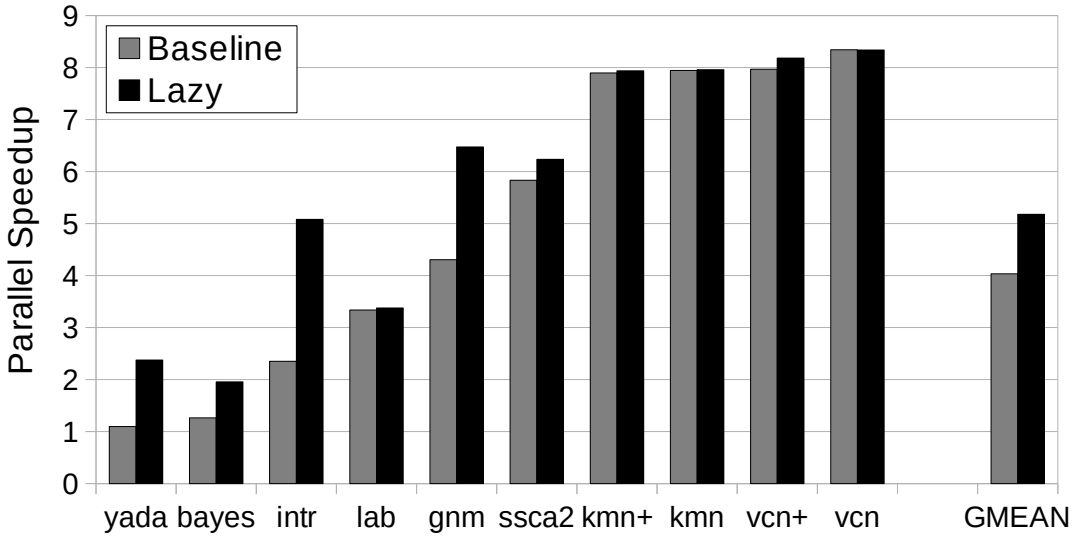



Figure 4.6: Overall Results

The overall performance improvement can be seen in Figure 4.6. Forgive-TM achieves around 50% performance improvement in bayes and genome, and almost 2.1x improvement in intruder and yada. Although there is limit improvement with kmeans, ssca2, and vacation, this is more due to the fact that there is very limited number of conflict aborts in these benchmarks in the first place. As a result, there is a 29% performance improvement overall shown in the sTAMP benchmark suite.

Figure 4.7 looks at the distribution of the types of aborts that were detected. *Eager/Eager* aborts are data conflict aborts between eagerly accessed (and detected) read and write operations. *Eager/Lazy* aborts are between an eagerly accessed data, and a lazily accessed

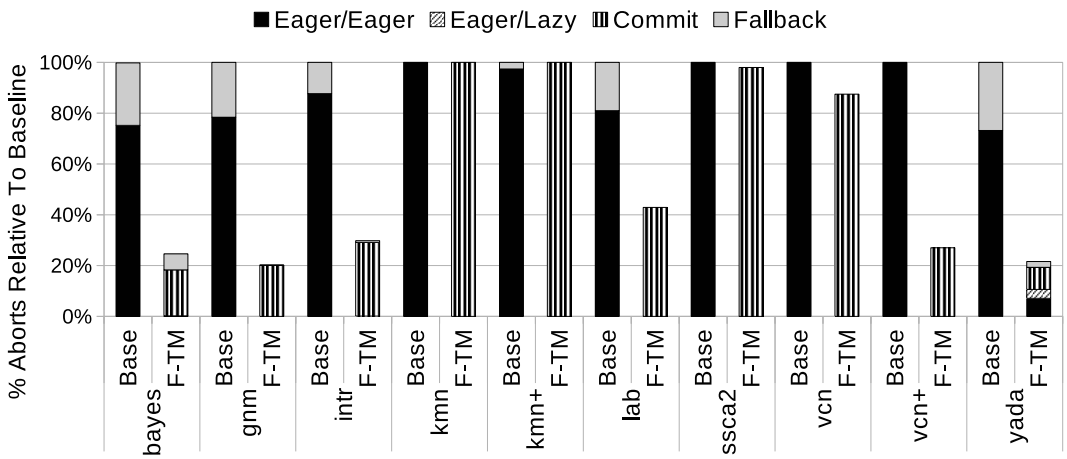
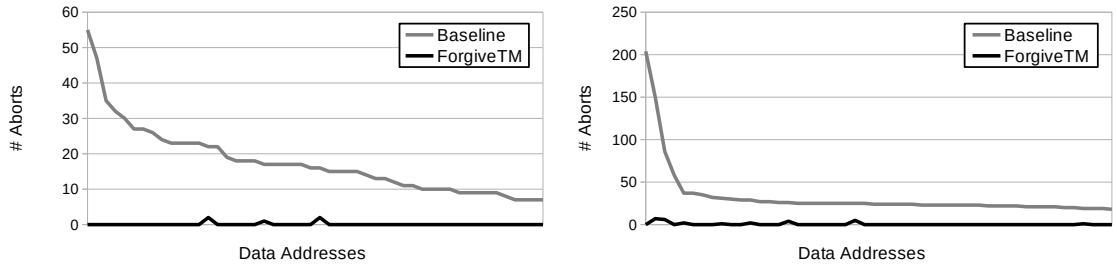



Figure 4.7: Abort Types

data. This commonly happens when a lazy write conflicts with an eager write (because the lazy write acts like an eager read), or a eager write conflicts with a lazy write. *Commit* aborts occur when a transaction with lazy writes start the *Commit-Prep* phase. For each entry within the LazySet the transaction issues a `GetM` request, which can lead to a data conflict with another transaction which accessed the same data. Obviously the baseline scheme will have no *Eager/Lazy* aborts or *Commit* aborts. *Fallback* aborts can occur because the fallback path was activated. This commonly occurs when a transaction is aborted 12 times due to a conflict abort (for capacity aborts the fallback path is taken immediately).

Each of these aborts were categorized, and then summed up and compared against the baseline scheme. The first thing that can be noticed is that lazy conflict detection can significantly reduce the number of aborts, in some cases up to a 5th. Another thing that can be noticed is that many of the data conflict aborts are converted from an eager abort to the lazy abort. In all but bayes and yada, all of the conflict abort were converted. This leads to the high performance gain that was shown in Figure 4.6.

In figure 4.8, we investigate in more detail how lazy conflict detection improves performance over the baseline configuration. We look at the histogram of the number conflict aborts due to eager writes. The trigger address for each conflict abort was counted, and




(a) # Eager Aborts By Each Address in Genome (b) # Eager Aborts By Each Address in Yada

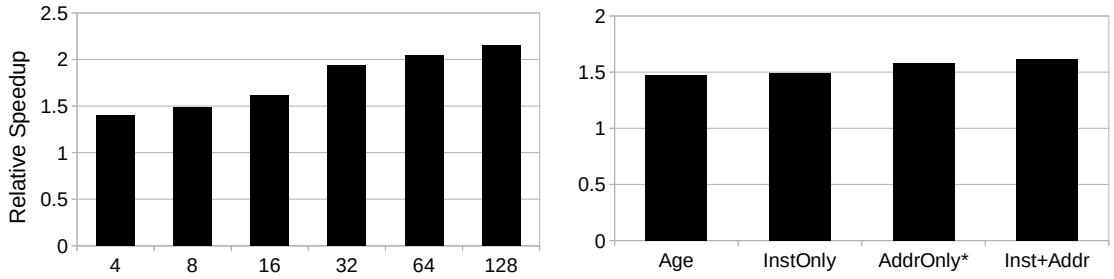
Figure 4.8: # Eager Aborts Triggered by Each Address
X axis is sorted by #aborts from baseline

look at the top 50 addresses, and yada in particular.

There are two things that can be noticed here. First, under the baseline configuration, there is significant skew in the number of aborts each address causes. In comparison, under Forgive-TM, the distribution of such aborts are much flatter. Some of the addresses are shifted over to lazy write aborts. However, as can be seen in Figure 4.7, the total number of aborts are reduced as well so this is not the entire explanation. Second, it also shows that the scoring mechanism does a good job in selecting only the addresses that can be problematic. After experiencing a few aborts, the system quickly learns which addresses are problematic and preemptively shifts them over to the lazy set. If the lazy set was less effective, this would result in addresses getting evicted and potentially triggering more eager aborts.

In Figure 4.9, we look do a sensitivity analysis of two aspects of the lazy set: the size of the lazy set, and the scoring mechanism. First, in Figure 4.9a, we look at how the size of the lazy set affects to overall performance. For each configuration we compute the average relative speedup over the baseline configuration. We exclude results from kmeans and vacation, since lazy writes do not have much of an impact.

As we can see, the larger the size of the lazy set, the more writes can be done lazily. Although the large lazy set can result in a longer *Commit-Prep* phase overhead, the aborts avoided by doing lazy writes can more than overcome the *Commit-Prep* overhead.



(a) LazySet Sizes (b) Scoring Policies

Figure 4.9: Sensitivity Analysis With the LazySet

s just been shown, having a larger lazy set is beneficial since more write instructions can be done lazily. However, some write operations are more "conflict-prone" than others [69]. Therefore, it makes sense to have a scoring policy where the lazy set discriminates writes and prefers certain write operations over others. This is where the scoring policies make a difference. In Figure 4.9b we compare the various scoring policies. We use a lazy set size of 16 for each.

The *Age* policy prefers older writes over younger writes. It does this adding all writes to the lazy set until the set is full, and never evicting any entry once it's added. This policy is the simplest, and acts as a baseline policy to any scoring scheme.

The *Inst* policy keeps track of the number of aborts each write instruction eventually triggers, and uses that number as its score. The idea is that store instructions that are likely to trigger data conflict aborts are likely to trigger them again.

When the data address for a given write instruction exists in the score table, the PC of the write instruction is saved in the instruction score table. Later, when a coherence request results in a data conflict abort, the PC is looked up and the corresponding entry is incremented. Later, when the instruction score is computed, the following equation is used:

$$InstScore = \begin{cases} 2 & \text{if num\_aborts is non-zero} \\ 0 & \text{if num\_aborts is zero} \end{cases}$$

The *Addr* policy looks at the number of aborts for each destination address, and is the

used in Forgive-TM. The idea is that a data address that triggers a lot of aborts, that is "hot," will continue to be popular. This is the one used in Forgive-TM, and is described in more detail in Section 4.3.3.

Last, the *Inst+Addr* policy uses the sum of the *InstScore* and the *AddrScore*. This policy tries to take the best of both the instruction-based scoring policy and the address-based scoring policy.

As can be seen in Figure 4.9b, there is benefit to be had with a more sophisticated scoring policies. Although there is some benefit by discriminating write operations based on the instruction, which is done by *Inst* and *Inst+Addr*, using the address score is more than enough to overcome any performance disadvantages due a small scoring policy. In addition, the instruction-based scoring policies require additional complexity by maintaining a table of write instructions to every cache line. As a result, in Forgive-TM we simply use the *Addr* policy.

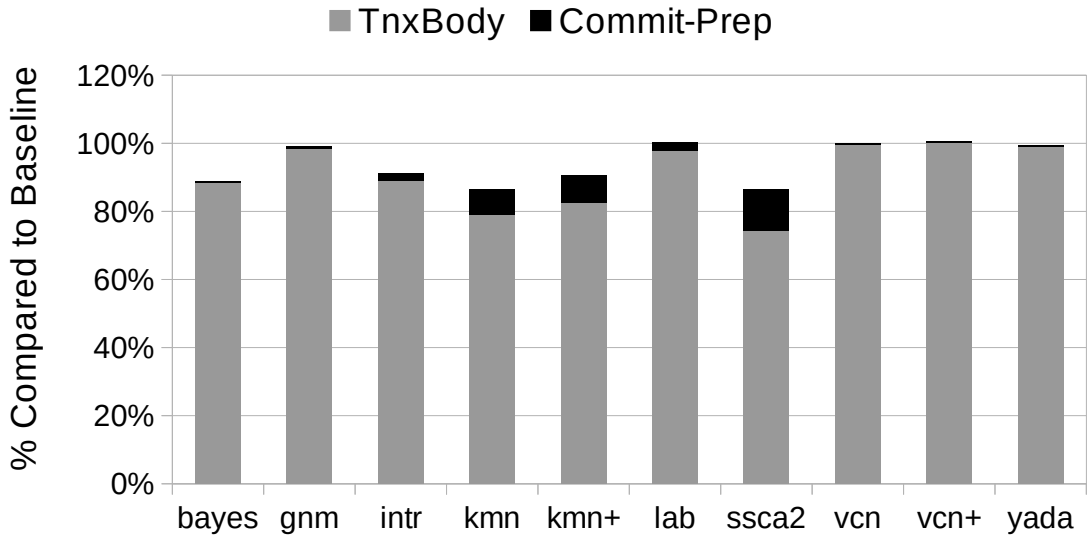



Figure 4.10: Ratio of Useful Work Compared to Baseline

One performance pathology of lazy conflict detection is the extended commit phase [71]. Whereas under eager conflict detection the write operations are executed as they are encountered, under lazy conflict detection, the operations are bunched together at the end

of the transaction. Therefore, it is important to make sure that the overhead due to the *Commit-Prep* phase is limited.

In figure 4.10, we look at the ratio of *useful work* done by Forgive-TM compared to baseline. Useful work in this case means the transaction body (transaction begin to the start of `Commit-Prep`) and the time spent in the `Commit-Prep` phase. The total cycles of these two phases are added up and compared against the total cycles from the transaction body in baseline.

As can be seen, in most cases this ratio is limited. Kmeans (both versions) and ssca2 do have a larger percentage than the others, though. This is because the transaction sizes in these benchmarks are very short.

Note however, that unlike TCC-style HTMs where commit arbitration can result in serialization of transaction commits, Forgive-TM relies on the conflict detection of the existing HTM hardware and thus can commit multiple transactions in parallel. A side effect of our mechanism is that even though a transaction may encounter a data conflict during the *Commit-Prep* phase and be aborted, however, this was rare in our experiments.

# CHAPTER 5
# CONCLUSION

Commercial HTM offerings are becoming common. However, the design decisions made to support HTM remain obstacles for good performance. First, many use a requester-wins style of conflict resolution. However, requester-wins can lead to suboptimal results when deciding which transaction to abort. Second, the transactions get aborted due to many reasons other than data conflicts, resulting in transactions that were still valid being aborted. Last, conflict detection is done eagerly, although it has been shown that lazy conflict detection tends to perform better.

In this thesis, we propose a set of modifications to improve the situation. PleaseTM is a mechanism that can provide greater freedom in conflict management, while still leaving the coherence protocol untouched. It uses plea bits that are inserted in coherence response messages as a simple payload, untouched by the coherence protocol. These bits are forwarded to the requester, and enable hardware to make more intelligent decisions about deciding which transaction to abort. PleaseTM, by allowing better conflict management, can provide a significant performance improvement compared to baseline requester-wins HTM, and can work in concert with a variety of software fallbacks.

Pre-abort handlers provide the programmer an opportunity to intervene when an abort-causing situation is encountered. When inside the non-transactional handler function, mitigating action can be taken to avoid the abort, or non-transactional work can be included, which has previously required custom hardware support. Pre-abort handlers allow support such additional functionality, through a simple addition to best-effort HTM.

Lastly, Forgive-TM emulates lazy conflict detection over conventional HTM, which uses eager conflict detection. Lazy conflict detection allows better performance by reducing the window of vulnerability and avoiding futile aborts. It does this by doing speculative

writes immediately, acquiring write permission later, just before the commit. This is when the data conflicts are detected, lazily.

Through these modifications, conventional HTMs are able to close the gap with more sophisticated HTMs, all while keeping the coherence protocol intact.

## REFERENCES


[1] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Intl. Symp. on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel, 2013, pp. 225–236, ISBN: 978-1-4503-2079-5.

[2] Intel Corporation, *Intel® architecture instruction set extensions programming reference*, 2012.

[3] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for ibm system z," in *Intl. Symp. on Microarchitecture (MICRO)*, ser. MICRO 45, 2012, pp. 25–36.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Intl. Symp. on Computer Architecture*, ser. ISCA '04, München, Germany, 2004, pp. 102–, ISBN: 0-7695-2143-6.

[5] Y. Liu, S. Diestelhorst, and M. Spear, "Delegation and nesting in best-effort hardware transactional memory," in *ACM Symp. on Parallelism in Algorithms and Architectures*, ser. SPAA '12, Pittsburgh, Pennsylvania, USA, 2012, pp. 38–47, ISBN: 978-1-4503-1213-4.

[6] S. Wasson, *Errata prompts intel to disable tsx in haswell, early broadwell cpus*, http://techreport.com/news/26911/errata-prompts-intel-to-disable-tsx-in-haswell-early-broadwell-cpus; accessed 15-Nov-2015.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, USA, 2009, pp. 157–168, ISBN: 978-1-60558-406-5.

[8] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado, 2013, 19:1–19:11, ISBN: 978-1-4503-2378-9.

[9] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc, "Xcalls: Safe i/o in memory transactions," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09, Nuremberg, Germany: ACM, 2009, pp. 247–260, ISBN: 978-1-60558-482-9.

[10] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[11] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Intl. Symp. on Computer Architecture*, ser. ISCA '07, San Diego, California, USA, 2007, pp. 81–91, ISBN: 978-1-59593-706-3.

[12] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "Logtm: Log-based transactional memory," in *Intl. Symp. on High-Performance Computer Architecture*, ser. HPCA '06, 2006, pp. 254–265.

[13] M. Lupon, G. Magklis, and A. Gonzalez, "Fastm: A log-based hardware transactional memory with fast abort recovery," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '09, 2009, pp. 293–302.

[14] Intel Corporation, *Intel® 64 and ia-32 architectures optimization reference manual*, 2014.

[15] Y. Liu and M. Spear, "Toxic transactions," in *6th Workshop on Transactional Computing*, Transact, 2011.

[16] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Symp. on Parallelism in Algorithms and Architectures*, ser. SPAA '08, Munich, Germany, 2008, pp. 169–178, ISBN: 978-1-59593-973-9.

[17] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *IEEE/ACM Intl. Symp. on Microarchitecture*, ser. MICRO 42, New York, New York, 2009, pp. 156–167, ISBN: 978-1-60558-798-1.

[18] G. Blake, R. Dreslinski, and T. Mudge, "Bloom filter guided transaction scheduling," in *Intl. Symp. on High Performance Computer Architecture*, ser. HPCA '11, 2011, pp. 75–86.

[19] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: Avoiding conflicts in transactional memories," in *ACM Symp. on Principles of Distributed Computing*, ser. PODC '09, Calgary, AB, Canada, 2009, pp. 7–16, ISBN: 978-1-60558-396-9.

[20] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for gpu architectures," in *IEEE/ACM Intl. Symp. on Microarchitecture*, ser. MICRO 44, Porto Alegre, Brazil, 2011, pp. 296–307, ISBN: 978-1-4503-1053-6.

[21] F. Tabba, A. W. Hay, and J. R. Goodman, "Transactional conflict decoupling and value prediction," in *Intl. Conf. on Supercomputing*, ser. ICS '11, Tucson, Arizona, USA, 2011, pp. 33–42, ISBN: 978-1-4503-0102-2.

[22] E. Akpinar, T. Saša, O. Unsal, A. Cristal, and M. Valero, "A comprehensive study of conflict resolution policies in hardware transactional memory," in *6th Workshop on Transactional Computing*, Transact, 2011.

[23] J. Poe, C.-B. Cho, and T. Li, "Using analytical models to efficiently explore hardware transactional memory and multi-core co-design," *Intl. Symp. on Computer Architecture and High Performance Computing,*, vol. 0, pp. 159–166, 2008.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, *SESC simulator*, http://sesc.sourceforge.net, 2005.

[25] O. Tange, "Gnu parallel - the command-line power tool," pp. 42–47, 2011.

[26] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for MultiProcessing," in *IEEE Intl. Symp. on Workload Characterization*, 2008, pp. 35–46.

[27] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *Intl. Conf. on Autonomic Computing*, USENIX Association, Jun. 2014, pp. 209–219, ISBN: 978-1-931971-11-9.

[28] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *Intl. Conf. on Supercomputing*, ser. ICS '09, Yorktown Heights, NY, USA, 2009, pp. 136–146, ISBN: 978-1-60558-498-0.

[29] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 4, 42:1–42:25, Dec. 2013.

[30] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8," in *Intl. Symp. on Computer Architecture*, ser. ISCA '15, Portland, Oregon, 2015, pp. 144–157, ISBN: 978-1-4503-3402-0.

[31] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence decoupling: Making use of incoherence," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI, Boston, MA, USA, 2004, pp. 97–106, ISBN: 1-58113-804-0.

[32] M.-M. Waliullah and P. Stenstrom, "Classification and elimination of conflicts in hardware transactional memory systems," in *Intl. Symp. on Computer Architecture and High Performance Computing*, ser. SBAC-PAD '11, 2011, pp. 96–103, ISBN: 978-0-7695-4573-8.

[33] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*, ser. PACT '14, Edmonton, AB, Canada, 2014, pp. 3–14, ISBN: 978-1-4503-2809-8.

[34] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *ACM Symp. on Principles of Distributed Computing*, ser. PODC '14, Paris, France, 2014, pp. 212–221, ISBN: 978-1-4503-2944-6.

[35] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, California, USA: ACM, 2007, pp. 24–34, ISBN: 978-1-59593-706-3.

[36] C. Zilles and R. Rajwar, "Transactional memory and the birthday paradox," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07, San Diego, California, USA: ACM, 2007, pp. 303–304, ISBN: 978-1-59593-667-7.

[37] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott, "Nonblocking transactions without indirection using alert-on-update," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '07, San Diego, California, USA: ACM, 2007, pp. 210–220, ISBN: 978-1-59593-667-7.

[38] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06, Washington, DC, USA: IEEE Computer Society, 2006, pp. 53–65, ISBN: 0-7695-2608-X.

[39] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "Asf: Amd64 extension for lock-free data structures and transactional memory," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 39–50.

[40] J. Chung, D. R. Chakrabarti, and C. C. Minh, "Analysis on semantic transactional memory footprint for hardware transactional memory," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010, pp. 1–11.

[41] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero, "Debugging programs that use atomic blocks and transactional memory," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10, Bangalore, India: ACM, 2010, pp. 57–66, ISBN: 978-1-60558-877-3.

[42] M. Wong and V. Luchangco, *Sg5 transactional memory support for c++ update*, 2014.

[43] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08, Munich, Germany: ACM, 2008, pp. 285–296, ISBN: 978-1-59593-973-9.

[44] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "Tokentm: Efficient execution of large transactions with hardware transactional memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 127–138, ISBN: 978-0-7695-3174-8.

[45] N. Nethercote, R. Walsh, and J. Fitzhardinge, ""building workload characterization tools with valgrind"," in *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 2–2.

[46] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir, "Pitfalls of lazy subscription," in *Workshop on the Theory of Transactional Memory*, ser. WTTM '14, Paris, France, 2014.

[47] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, ser. PODC '03, Boston, Massachusetts: ACM, 2003, pp. 92–101, ISBN: 1-58113-708-7.

[48] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Rivière, "Deadline-aware scheduling for software transactional memory," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011, pp. 257–268.

[49] S. Diestelhorst, M. Nowack, M. Spear, and C. Fetzer, "Brief announcement: Between all and nothing - versatile aborts in hardware transactional memory," in *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '13, Montréal, Québec, Canada: ACM, 2013, pp. 108–110, ISBN: 978-1-4503-1572-2.

[50] L. Baugh and C. Zilles, "An analysis of i/o and syscalls in critical sections and their implications for transactional memory," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, 2008, pp. 54–62.

[51] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: ACM, 2006, pp. 336–346, ISBN: 1-59593-451-0.

[52] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06, New York, New York, USA: ACM, 2006, pp. 209–220, ISBN: 1-59593-189-9.

[53] I. Calciu, J. Gottschlich, T. Shpeisman, M. Herlihy, and G. Pokam, "Invyswell: A hybrid transactional memory for haswell's restricted transactional memory," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014, pp. 187–199.

[54] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011. eprint: https://doi.org/10.2200/S00346ED1V01Y201104CAC016.

[55] T. Harris, J. Larus, and R. Rajwar, "Transactional memory, 2nd edition," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010. eprint: https://doi.org/10.2200/S00272ED1V01Y201006CAC011.

[56] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06, Washington, DC, USA: IEEE Computer Society, 2006, pp. 227–238, ISBN: 0-7695-2608-X.

[57] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm: Eager-lazy hardware transactional memory," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, New York, New York: ACM, 2009, pp. 145–155, ISBN: 978-1-60558-798-1.

[58] J. G. Beu, J. A. Poovey, E. R. Hein, and T. M. Conte, "High-speed formal verification of heterogeneous coherence hierarchies," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 566–577.

[59] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 97–108.

[60] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, San Diego, California, USA: ACM, 2007, pp. 278–289, ISBN: 978-1-59593-706-3.

[61] X. Qian, W. Ahn, and J. Torrellas, "Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, Washington, DC, USA: IEEE Computer Society, 2010, pp. 447–458, ISBN: 978-0-7695-4299-7.

[62] X. Qian, J. Torrellas, B. Sahelices, and D. Qian, "Bulkcommit: Scalable and fast commit of atomic blocks in a lazy multiprocessor environment," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, Davis, California: ACM, 2013, pp. 371–382, ISBN: 978-1-4503-2638-4.

[63] A. Negi, R. Titos-Gil, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Pi-tm: Pessimistic invalidation for scalable lazy hardware transactional memory," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[64] A. Negi, M. M. Waliullah, and P. Stenstrom, "Lv*: A class of lazy versioning htms for low-cost integration of transactional memory systems," in *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies*, ser. IFMT '10, Saint-Malo, France: ACM, 2010, 5:1–5:10, ISBN: 978-1-4503-0008-7.

[65] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "Si-tm: Reducing transactional memory abort rates through snapshot isolation," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, Salt Lake City, Utah, USA: ACM, 2014, pp. 383–398, ISBN: 978-1-4503-2305-5.

[66] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *2008 International Symposium on Computer Architecture*, 2008, pp. 139–150.

[67] M. Lupon, G. Magklis, and A. Gonzalez, "A dynamically adaptable hardware transactional memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, Washington, DC, USA: IEEE Computer Society, 2010, pp. 27–38, ISBN: 978-0-7695-4299-7.

[68] R. Titos, M. E. Acacio, and J. M. Garcia, "Speculation-based conflict resolution in hardware transactional memory," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[69] L. Zhao, W. Choi, and J. Draper, "Sel-tm: Selective eager-lazy management for improved concurrency in transactional memory," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 95–106.

[70] S. Park, M. Prvulovic, and C. J. Hughes, "Pleasetm: Enabling transaction conflict management in requester-wins hardware transactional memory," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 285–296.

[71] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 81–91, Jun. 2007.

[72] W. Ruan, Y. Liu, and M. Spear, "Stamp need not be considered harmful," in *TRANSACT: 9th ACM SIGPLAN Workshop on Transactional Computing*, 2014.