PleaseTM: Enabling Transaction Conflict Management in Requester-wins Hardware Transactional Memory

Sunjae Park
Georgia Institute of Technology
sunjae.park@gatech.edu

Milos Prvulovic
Georgia Institute of Technology
milos@cc.gatech.edu

Christopher J. Hughes
Intel
christopher.j.hughes@intel.com

ABSTRACT

With recent commercial offerings, hardware transactional memory (HTM) has finally become an important tool in writing multithreaded applications. However, current offerings are commonly implemented in a way that keep the coherence protocol unmodified. Data conflicts are recognized by coherence messages sent by the requester to sharers of the cache block (e.g., a write to a speculatively read line), who are then aborted. This tends to abort transactions that have done more work, leading to suboptimal performance. Even worse, this can lead to live-lock situations where transactions repeatedly abort each other.

In this paper, we present PleaseTM, a mechanism that allows more freedom in deciding which transaction to abort, while leaving the coherence protocol design unchanged. In PleaseTM, transactions insert plea bits into their responses to coherence requests as a simple payload, and use these bits to inform conflict management decisions. Coherence permission changes are then achieved with normal coherence requests. Our experiments show that this additional freedom can provide on average 43% speedup, with a maximum of 7-fold speedup, on STAMP benchmarks running at 32 threads compared to requester-wins HTM.

1. INTRODUCTION

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 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.

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

978-1-4673-9211-2/16/$31.00

In this paper, 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.

Sending a plea does not guarantee survival for a transaction. Since the coherence protocol is unchanged, the responding core will still downgrade or invalidate the requested line. The core, after sending its plea-enhanced response, attempts to re-acquire the block in its original coherence state and validate that the data is identical to the original version. If it is, the responder knows that the work done so far is not invalid and can continue. If not, this means there was an intervening memory operation (violating atomicity) and the transaction needs to abort.

Conflict resolution using these plea bits does not affect the transactions' correctness. HTMs provide *strict isolation*, where transactions appear to happen instantaneously, at commit. This requires a consistent read set and for the writes to become visible all at once, relative to other transactions and non-transactional operations. Refetching and validating blocks that incur conflicts is consistent with this requirement [7].

Using the plea bits, we can keep the coherence protocol unmodified, while enabling a rich set of conflict resolution policies, many of which are superior to the default requester-wins approach in current best-effort HTMs. Our evaluation results indicate that PleaseTM can result in significant performance improvement even by simply reversing the requester-wins policy, and additional benefits can be obtained by following more sophisticated conflict resolution policies enabled by PleaseTM. On a simulated multicore system running STAMP benchmarks with 32 threads, an implementation that adds plea bits to coherence responses has an average speedup of 43%, with a maximum speedup of 7× over a requester-wins baseline.

This paper is organized as follows: Section 2 explains how requester-wins HTM leads to performance problems due to wasted work. Section 3 outlines the basic version of our proposal by using a simple 1-bit plea to swap the abort decision when transactions conflict. Section 4 looks at an enhancement to our proposal to support a wide variety of conflict resolution policies by using more plea bits instead of a single-bit plea, and we discuss correctness of doing conflict resolution using our proposal in Section 5. In Section 6 we explain our simulation setup and show experimental results. We cover related work in Section 7 and then conclude the paper.

2. RATIONALE

2.1 Requester-Wins HTM Conflict Handling

Best-effort, requester-wins HTMs detect data conflicts via coherence requests [7]. Figure 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 message 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.

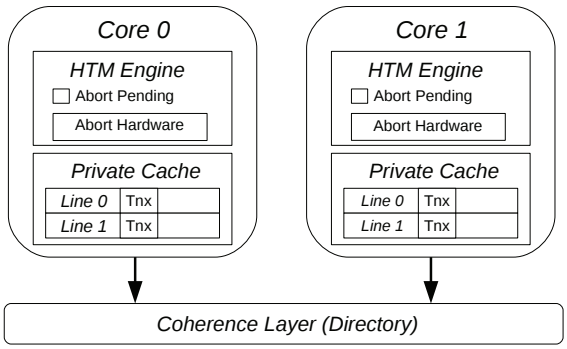


Figure 1: Baseline Requester-wins HTM Architecture

This type of best-effort HTM has been the dominant mechanism in providing TM support for commercial offerings [1, 8, 2, 3, 9]. 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 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).

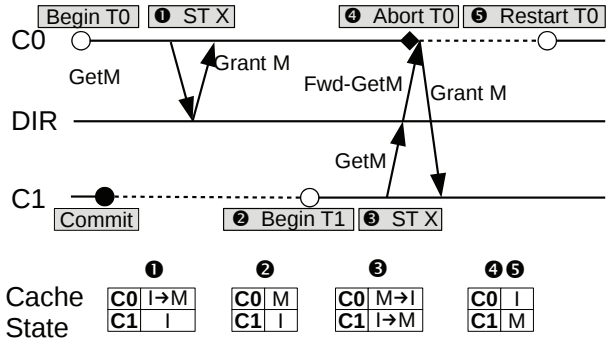


Figure 2: Timeline of wasted work in requester-wins HTM

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, 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 [10].

2.2 Prior Solutions

If we allow modification to the coherence mechanism, there are ways to reduce the impact of these issues. For example, in LogTM [11], 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 [12] 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 [13, 14, 15]. 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 [16, 17, 18]).

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.

3. PLEASETM

3.1 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[2]. 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). 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 [19, 20]. 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* [7], 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* [7].

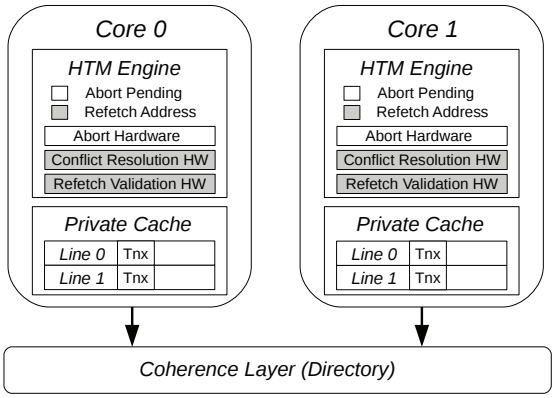


Figure 3: Architecture for PleaseTM

Figure 3 shows the architecture for PleaseTM, with the additional components shown 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.

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

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

3.2 Write-write Conflict with Plea Accepted

Returning to our example, Figure 4 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).

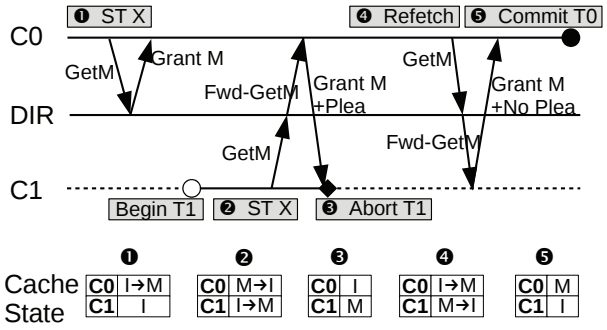


Figure 4: Write-write conflict with T0's plea accepted

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).

3.3 Write-write Conflict with Plea Denied

In the previous example, T0 avoided an abort, but this won't always happen. Figure 5 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).

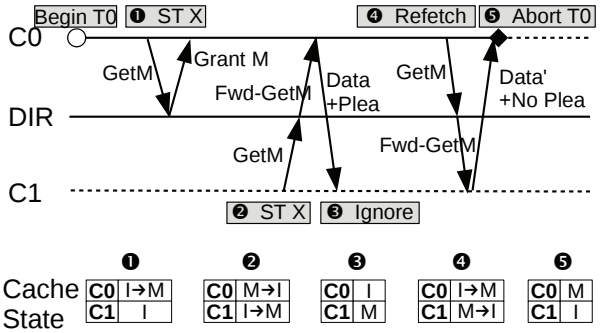


Figure 5: Write-write conflict with T0's plea denied

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.

3.4 Read-write Conflict with Plea Accepted

Figure 6 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).

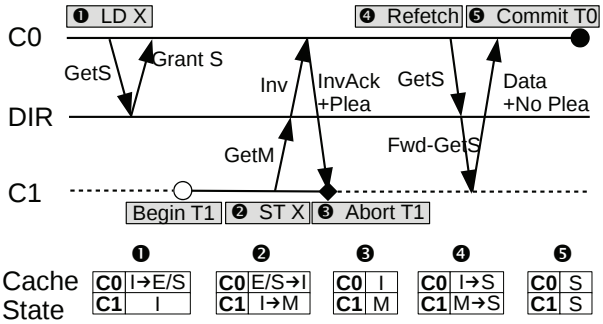


Figure 6: Read-write conflict with plea accepted

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 denied, just like in Section 3.3. Similarly, multiple readers are aborted after they refetch, as all their copies are now stale.

4. 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 [21, 12] 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.

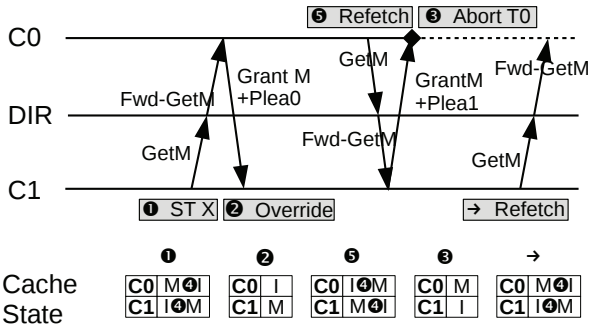


Figure 7: A write-write conflict with a different conflict resolution policy

In PleaseTM, we can expand the plea to include this information instead. Figure 7 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 6.3.

5. 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).

5.1 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. Follow-

ing 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 8 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.

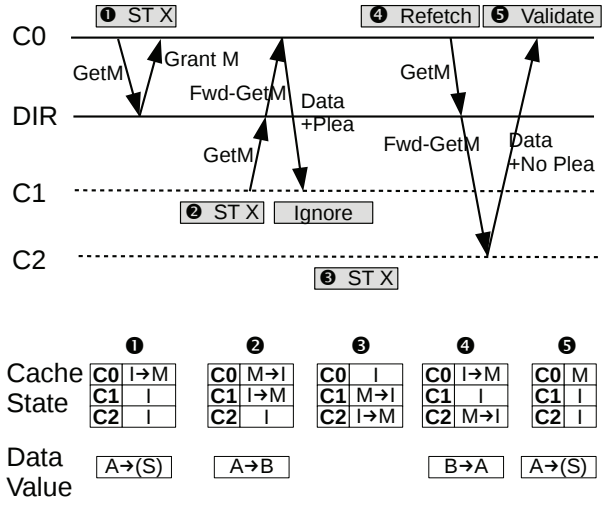


Figure 8: Timeline for A → B → A

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 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.

6. EXPERIMENTAL RESULTS

6.1 Setup

We modify SuperTrans [22], a transactional memory simulator built from SESC [23], 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 [24].

We evaluate the benefits of our approach using STAMP benchmarks [25] 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 [13].

6.2 Overall Results

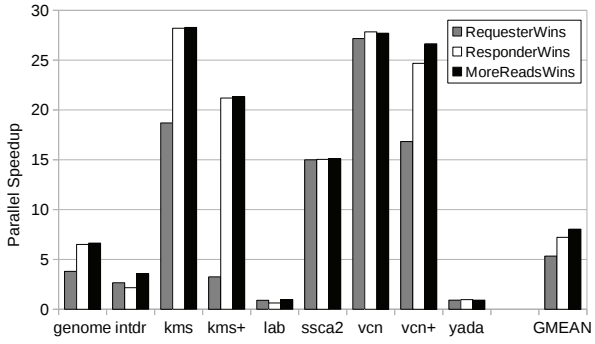


Figure 9: Baseline (*RequesterWins*) vs. Two Simple Plea-enabled Policies

Figure 9 shows the parallel speedup (relative to the single-threaded run) for several conflict-resolution policies. The first policy shown is the baseline *RequesterWins* policy used in best-effort HTMs, where the transaction requesting the block "wins," and the transaction(s) that receives the invali-

dation/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.

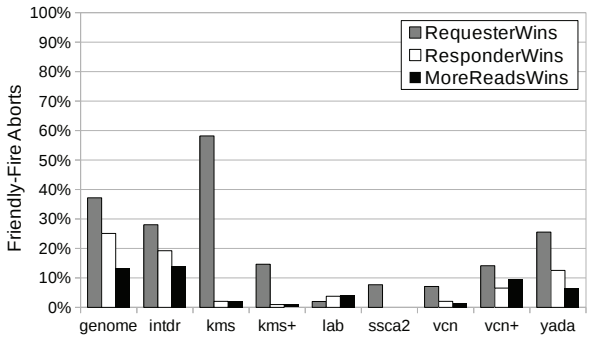


Figure 10: Ratio of Friendly-fire Aborts

Figure 10 shows the percentage of friendly-fire aborts [10] 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, 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.

6.3 More Conflict Resolution Policies

As shown previously [21, 12], 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 11 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

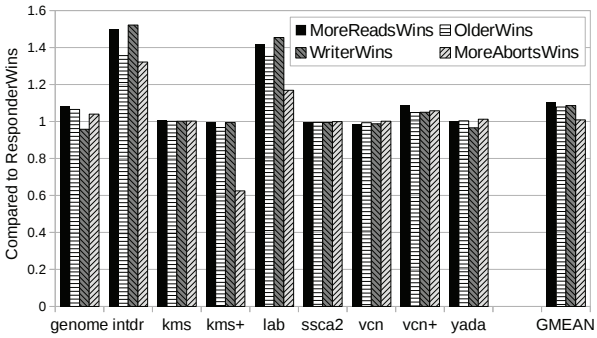


Figure 11: Speedup of Various Types of Plea Bits Compared to *ResponderWins*

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.

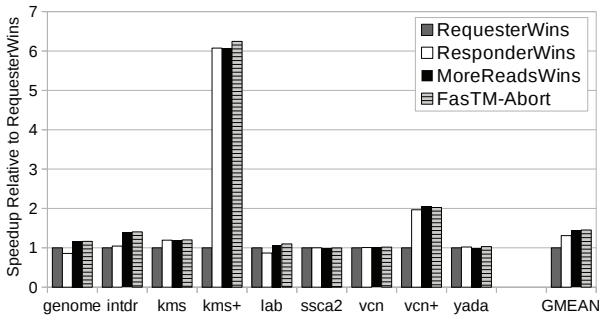
WriterWins largely performs similar to *MoreReadsWins*, but performs worse in genome. This is because we always favor the writer, even if the requester sends a `GetM` request and conflicts with multiple readers; each of which might have done more work.

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.

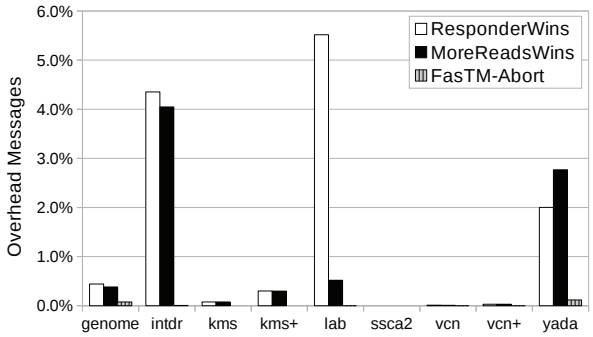
6.4 Overhead of Avoiding Coherence Protocol Modifications

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.

In figure 12, we show the overhead due to these additional messages. Figure 12a 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 12b shows the number of overhead messages as a per-



(a) Performance Overhead of Avoiding Coherence Modifications



(b) Overhead Messages (Refetch/Nack)

Figure 12: Overhead of Avoiding Coherence Modifications

centage 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 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.

6.5 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 [13].

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 critical section to execute in isolation. Aborting all these transactions can have a perfor-

mance 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.

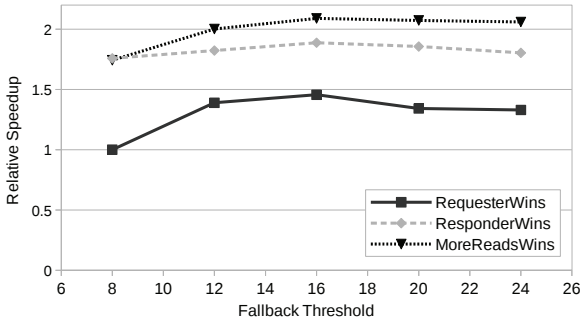


Figure 13: Speedup With Different Fallback Thresholds

In figure 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 [9, 26]. With the threshold being too large, conflicting transactions may simply continue to abort each other without increasing the probability of success. However, as we can see from figure 10, *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."

6.6 Conflict Resolution and Software Fallback

Our baseline system uses a global lock fallback, as described in Section 6.1. 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 [27]. 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 [14]. 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 [28]. 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.

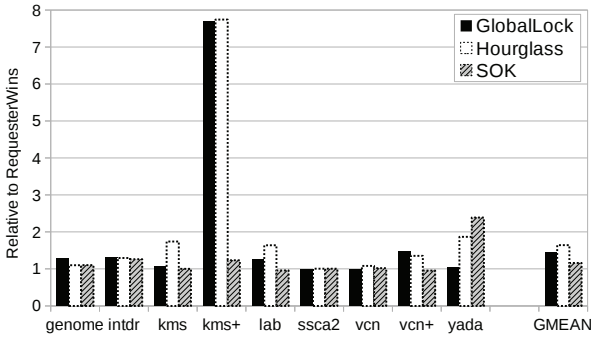


Figure 14: Speedup Chart Various Software Fallbacks

Figure 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*.

6.7 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. [29]

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.

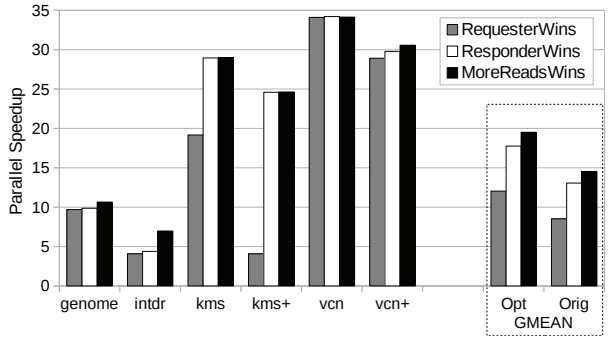


Figure 15: Speedup Chart with Optimized Software

Figure 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*.

7. RELATED WORK

LogTM is an HTM scheme that detects data conflicts using a modified directory-based coherence protocol [11]. 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 [12] 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. [21] and Shriraman et al. [27] 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 [30, 20] 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 [31] 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. [9] investigate the performance Intel's best-effort HTM system and briefly discuss how to tune for performance. Diegues and Romano [26] look at tuning parameters such as the fallback threshold at runtime using machine learning. Diegues et al. [32] 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 [15] 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 [14] 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 6.6. Software-assisted Conflict Management [33] 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. [28] look at several proposals that serialize transactions so that restarted transactions happen one after another. Proactive Transaction Scheduling [16] 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 [17] uses Bloom filters of the read and write sets to inform the conflict predictor. Preventing versus Curing [18] instead uses the past history of the read and write operations to predict the transactions' access set to predict conflicts.

8. CONCLUSION

Among the commercial offerings of hardware transactional memory (HTM), many use a requester-wins style of conflict resolution. However, requester-wins can lead to suboptimal results when deciding which transaction to abort. Despite this disadvantage, this has been the dominant style of HTM because the coherence protocol can be kept unmodified.

We propose PleaseTM, mechanism that can provide greater freedom in conflict management, while still leaving the coherence protocol untouched. Our mechanism 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. We show that 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.

9. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1320717. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

10. 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*, ISCA '13, pp. 225–236, 2013.

[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)*, MICRO 45, pp. 25–36, Dec 2012.

[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*, ISCA '04, pp. 102–, 2004.

[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*, SPAA '12, pp. 38–47, 2012.

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

[8] 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*, ASPLOS XIV, pp. 157–168, 2009.

[9] 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*, SC '13, pp. 19:1–19:11, 2013.

[10] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Intl. Symp. on Computer Architecture*, ISCA '07, pp. 81–91, 2007.

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

[12] 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*, PACT '09, pp. 293–302, Sept 2009.

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

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

[15] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Symp. on Parallelism in Algorithms and Architectures*, SPAA '08, pp. 169–178, 2008.

[16] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 42, pp. 156–167, 2009.

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

[18] 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*, PODC '09, pp. 7–16, 2009.

[19] 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*, MICRO 44, pp. 296–307, 2011.

[20] F. Tabba, A. W. Hay, and J. R. Goodman, "Transactional conflict decoupling and value prediction," in *Intl. Conf. on Supercomputing*, ICS '11, pp. 33–42, 2011.

[21] 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.

[22] 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.

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

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

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

[26] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *Intl. Conf. on Autonomic Computing*, pp. 209–219, USENIX Association, June 2014.

[27] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *Intl. Conf. on Supercomputing*, ICS '09, pp. 136–146, 2009.

[28] 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, pp. 42:1–42:25, Dec. 2013.

[29] 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*, ISCA '15, pp. 144–157, 2015.

[30] 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*, ASPLOS XI, pp. 97–106, 2004.

[31] 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*, SBAC-PAD '11, pp. 96–103, 2011.

[32] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT '14, pp. 3–14, 2014.

[33] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *ACM Symp. on Principles of Distributed Computing*, PODC '14, pp. 212–221, 2014.