Forgive-TM: Supporting Lazy Conflict Detection In Eager Hardware Transactional Memory

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

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

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

***Abstract*—Commercial hardware transactional memory (TM) systems commonly use coherence messages to detect data conflicts. When a core inside a transaction receives a coherence request for data, it uses this information to determine whether there was a data conflict. Inherent in this behavior is the fact that data conflicts are detected eagerly, i.e., as soon as possible, and even while both sides of the conflict are speculative. Although it has been shown that lazy conflict detection can lead to better performance, this approach precludes lazy detection.**

In this paper, we describe a mechanism that allows conventional hardware to support lazy conflict detection, while still keeping the coherence protocol intact. Under Forgive-TM, speculative writes are done immediately to a special buffer, without first obtaining global write permission. The write permission is acquired later, when the transaction is about to commit. In other words, it "acts first, and asks forgiveness later." This effectively allows conflict detection to be done lazily. Using this scheme, ForgiveTM is able to provide 19% overall performance improvement in STAMP.

***Index Terms*—Parallel processing, Multithreading**

I. Introduction

The number of cores in modern processors is increasing. For multithreaded applications to benefit from them, they must effectively use multiple cores even in situations that require concurrency control. Transactional memory (TM), which allows multiple threads to optimistically access shared data concurrently, has been introduced as a way of improving performance scaling beyond that provided by the traditional approach based on mutex locks, which pessimistically serializes such accesses.

Hardware TM (HTM) provides support for TM in the hardware. Recently there has been renewed interest in HTM as several hardware vendors have implemented it in their processors [1]–[4]. However, unlike many of the designs proposed by the research community, the capabilities provided by these conventional HTMs are comparatively limited.

One major difference between commercial and research approaches is in how data conflicts are detected. A data conflict occurs when two or more concurrent transactions access the same data, and at least one of them is a write. HTMs detect conflicts either eagerly or lazily. With eager detection, the conflict between a pair of accesses is detected when the second access in the pair occurs. With lazy detection, a conflict is detected when one of the conflicting transactions attempts to commit. Either way, the conflict must be resolved, e.g., by aborting one or more conflicting transactions.

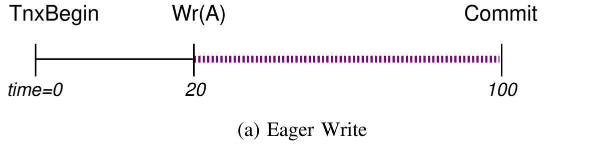
There is strong evidence that lazy conflict detection tends to yield better performance [5]. One of the reasons for this is futile transaction aborts, where an aborted transaction can restart and abort the aborter [6]. In eager conflict detection, speculative accesses are exposed to other threads as soon as they occur, thus creating a "window of vulnerability" during which a conflict with other threads may be detected (and acted upon), perhaps futilely. In contrast, lazy conflict detection occurs not long before a transaction commits, thus creating a very brief window of vulnerability.

This is illustrated in Figure 1, which shows the timeline of a transaction that writes a shared variable `A`, with eager and then with lazy conflict detection. In each timeline, the operations are shown above the line and, to provide perspective, example time stamps are shown below the line.

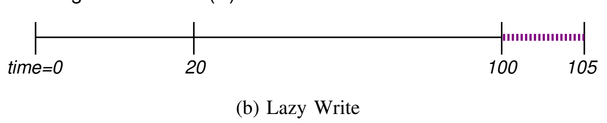
With eager detection (Figure 1a), the window of vulnerability begins as soon as the access occurs at cycle 20, and it ends at cycle 100 when the transaction commits, yielding an 80-cycle window of vulnerability. In contrast, with lazy detection (Figure 1b), the access also occurs at time 20, but its potential conflicts are not considered until later when, in cycle 100, the transaction starts preparing to commit. Thus the window of vulnerability begins only at cycle 100, and ends soon afterwards, when the transaction actually commits in cycle 105, yielding a 5-cycle window of vulnerability.

Although lazy conflict detection can significantly reduce the window of vulnerability, and thus the number of futile transaction aborts and other conflict-resolution actions, commercial HTMs mainly use eager conflict detection.

This is because they are designed to take advantage of the existing coherence protocol [7]. Specifically, a conflict is



(a) Eager Write



(b) Lazy Write

Fig. 1: Window of Vulnerability

detected when a `GetS` coherence request, caused by a load on another core, reaches a cache that has the same block in a transactionally-written (modified) state, or when a `GetM` request (from a store operation on another core) reaches a cache that has the same block in a transactionally-accessed (written or read) state. Since `GetS` and `GetM` messages are sent as soon as the access on another core occurs, conflict detection that relies only on these existing coherence messages has been eager.

In comparison, past proposals for lazy conflict detection have required additional coherence message types. Some proposals buffer writes locally and broadcast write sets at commit [8], [9]. This requires messages to first acquire commit permission, and then messages to broadcast the transaction's write set. Other proposals forgo the write set broadcast, but still retain the commit arbitration message [9]–[11], or do away with the commit arbitration, but instead add explicit abort request messages [5], [12], [13].

Unfortunately, additional coherence message types or coherence states complicate the design [7], [14]. Every additional message type and event results in an explosion in the state space [15]. In addition, because state transition requests are not atomic, high performance systems require coherence protocols with many more transient states [16], which make the problem even worse. As an example, a MOESI protocol, which is described with 5 states, is implemented in the Gem5 simulator with over 60 states (both stable and transient) [17], [18].

As a result, even commercially released processors can have bugs in their coherence protocol implementation [17], [19]. Formally verifying non-trivial coherence protocols for modern processors becomes intractable, failing after exceeding even 200GB of memory [20]. Randomized testing can help identify bugs in the design [21], but does not provide complete coverage. In-situ testing may help [17], but may not be commercially acceptable. The community therefore has recently pushed for simpler protocols [22], [23].

This motivation to deploy simple coherence protocols exists, even without TM. The bar is high for complicating coherence, which is arguably a big reason that lazy conflict detection has not been commercially deployed. Thus, we explore the question: can we provide effective lazy conflict detection without modifying the coherence protocol? To that end, we propose Forgive-TM. Forgive-TM provides a form of lazy conflict detection without needing additional message types such as commit arbitration request messages, write set broadcast messages, or abort request messages.

It does so building on top of an existing best-effort HTM. A transactional store operation is split into two parts: (1) a local, speculative, write, and (2) obtaining global write permission for the cache line. Forgive-TM tracks which local writes have not yet obtained write permission. At the end of the transaction, the transaction starts a new phase, called the `Commit-Prep` phase where it (belatedly) sends a `GetM` request to acquire global write permission. However, there is no need to coordinate the commit with the other processors, since the original coherence states and the original best-effort HTM hardware already provide notification (detection) if a conflicting access occurs at the same time. Once all the necessary write permissions are acquired, the transaction initiates the same `Commit` process as the original HTM design.

Forgive-TM does not affect cache coherence because the coherence requests and state transitions are done without any changes; the only changes are in the *timing* of a core sending conventional coherence requests. This delaying of coherence permission requests is still correct from the transaction's point of view, because the operations done by a transaction must be observed to be atomic [24]. Since dirty data from transactions is never made globally visible until the transaction commits, there is no externally visible difference in a transaction's behavior other than the precise timing of when it requests write permission, and possibly re-ordering of requests.

The rest of the paper is organized as follows: Section II explains in more detail how conflict detection works, and compares eager conflict detection with lazy conflict detection. Section V looks at prior work on this topic, and discusses how these proposals differ from ours. Section III introduces Forgive-TM, which allows conflicts to be detected lazily while keeping most of the baseline architecture intact. Section IV discuss our results, and we wrap up in Section VI.

II. HTM Conflict Detection

While a transaction is running, it needs to make sure it does not violate atomicity by leaking speculative data from itself or other transactions before it commits. Under requester-wins conventional HTMs, the data conflicts are detected through coherence messages [1]–[4].

In a conventional commercial HTM implementation, when a transaction speculatively writes to a cache line, it sends a `GetM` coherence request message, to obtain the most up-to-date contents of the line and write permission for that line from the directory, or other coherence mechanism [16], [24]. 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 before the first transaction committed and, to ensure transaction atomicity, that first transaction is forced to abort.

Likewise, if a transaction speculatively reads a cache line, it initially sends a `GetS` request, to obtain an up-to-date copy of the line, possibly in read-only (i.e., shared) state. Later, if it receives a `GetM`, which means that another thread is writing to the same line, the transaction is forced to abort to avoid violating atomicity.

Figure 2a depicts what happens as an example. There are two transactions, and the left-hand transaction first issues a speculative read (①). Later, when the righthand transaction issues a speculative write (②), its core sends a `GetM` message (③), which is later received by the first core. This notifies the lefthand transaction that it conflicted with another core, which then aborts itself (④).

As can be seen, the transaction is notified immediately of a potential data conflict, i.e. conflict detection is done *eagerly*. The advantage of this mechanism is that it needs no support

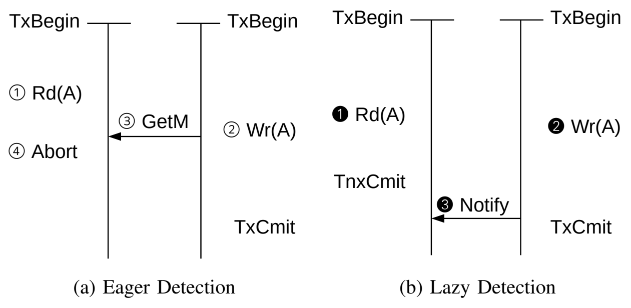


Fig. 2: Conflict Detection

from the coherence protocol: from the protocol's point of view, the core has executed ordinary reads and writes.

In contrast, there are TM systems that do *lazy* conflict detection instead. Under lazy detection, data conflicts are checked for only at the end of a transaction, when the transaction is ready to commit. If a transaction has a consistent set of inputs, and can obtain write permission for all output locations, the TM system can commit the transaction by making all speculative writes globally visible atomically.

Figure 2b depicts transactions with the same shared memory accesses as Figure 2a, but with lazy detection. As before, we have two transactions, and each issues speculative reads and writes (❶, ❷). Note that at this point, neither transaction notices a potential conflict because the righthand transaction buffers its write locally.

Later, the righthand transaction is ready to commit. The transaction starts the commit phase and notifies the lefthand transaction that it will be writing to A, which may trigger a 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.

Lazy conflict detection potentially allows much better performance compared to eager conflict detection for a number of reasons [5]. As shown in the examples above, lazy conflict detection delays the notification memory accesses that may lead to potential conflicts, and allows transaction ordering that can lead to better scalability. In addition, the window of vulnerability, shown in Figure 1 is reduced, avoiding unnecessary conflict aborts. Last, eager conflict detection can lead to futile aborts and livelocks [6], further impacting performance.

Broadly, there are three styles that have been proposed for implementing lazy conflict detection. First is the TCC family of HTMs [25]–[29]. These HTMs buffer all writes locally until commit time, not telling other cores (e.g., via coherence actions) that the writes have happened speculatively. When a transaction needs to commit, commit permission is requested. This is typically done via a dedicated central agent or bus that commercial HTMs do *not* include, and with new coherence/on-die messages. Once this has been granted, the write set is broadcast. Other transactions compare their read set with this and abort if there is a conflict, since they previously read now-stale data. Different members of this HTM family use different mechanisms for commit permission arbitration and write set broadcasting.

The second is the EazyHTM family [5], [12], [13]. These modify the coherence protocol to include message types which are used to construct local conflict sets. In other words, potential conflicts 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 frequency of such messages.

The third category builds on LogTM [9]–[11] and "lazifies" it. Instead of issuing transactional writes immediately (and storing the original version of the data in an undo log), these HTMs initially buffer the write internally. At commit time, transactions arbitrate for commit permission using an explicit commit request so that the commiting transaction has exclusive access. Once commit permission is acquired, transactions do a rapid broadcast of lazily written data with a special message type that indicates that the data is part of a committing transaction. This allows the commiting transaction to "win" any conflicts with other sharers. The other sharers, upon losing the data conflict, are forced to wait or get aborted.

Coherence protocols are already difficult to design even without the burden of HTM. As mentioned earlier, a real implementation of a coherence protocol may have many intermediate/transitory states, making correct implementation and verification difficult. Thus, changes to these designs must pass a very high bar.

Unfortunately, prior proposals that implement lazy conflict detection can lead to significant changes in the coherence protocol. Table I summaries the past proposals and the coherence protocol changes required.

TABLE I: Changes Required for Lazy Conflict Detection

	CommitReq	**WriteSet**	**ConfPolicy**	**AbortReq**
TCC	Yes	Yes	No	No
LazyLogTM	Yes	No	Yes	No
EazyHTM	No	No	No	Yes
ForgiveTM	No	No	No	No

TCC-style of HTMs need a mechanism to arbitrate for commit permission, and a mechanism for sending the transaction's write set to other cores that it may have a data conflict with. Both of these are additional coherence message types that need to be validated, increasing validation costs. In addition, these additional message types can be more complicated to validate than conventional message types. Conventional coherence messages work on individual cache lines: e.g. a given cache line is owned by a single core. In comparison, these message types work with sets of cache lines (the write set).

Lazified LogTM proposals build on an already elaborate base of LogTM, which introduces special "sticky" coherence states and NACK messages, which are specific to TM and are an additional burden to designers of the coherence protocol. They also support sophisticated conflict resolution policies, which determine which transaction "wins" a data conflict and is allowed to continue. Depending on the outcome, different response messages must be sent. A correctly implemented coherence protocol must properly validate all of these cases. In addition, these proposals use commit arbitration, which also affects conflict resolution outcomes.

EazyHTM style of HTMs also require changes. First, there are transactional request messages. These message types are similar to the conventional message types (like `GetS`), but the messages also indicate that the sender is inside of a transaction. At the end of a transaction, the processor sends an explicit abort message (another new coherence message type) to its peers. Lastly, the processors also keep track of additional transactional sets such as the killer set. The killer set indicates which processors have active transactions that may potentially conflict with the current processor's transaction. Cores now need to track sets of lines that other cores have accessed, and this carries a significant cost. The protocol now needs to be aware of transactions, and the associated sets of addresses.

As an alternative to previous approaches, we propose Forgive-TM, which allows mostly-lazy conflict detection on conventional HTMs, without changing the existing coherence layer.

III. Forgive-TM

In comparison to the prior work on lazy conflict detection, Forgive-TM is a mechanism that *allows* support for delaying data conflict detection *without* modifications to the coherence protocol. The changes are limited to the existing TM hardware, which lies within the core itself. This allows hardware vendors to harness the performance advantage of lazy conflict detection with only incremental hardware changes.

Forgive-TM's approach is to "act first, ask forgiveness later [1]." It performs a speculative write even if the core only has read permission for the line, and write permission is acquired only later, when the transaction is about to commit. This is allowed because, as long as all read and write operations from a transaction collectively "appear" to happen instantaneously, the operations can be rearranged [24]. This does not mean conflict detection is not done at all, and transactions that read from or write to shared data are notified of any other writers and will still be aborted.

In detail, Forgive-TM divides speculative writes into two categories: writes done eagerly and writes done lazily. The hardware chooses to perform some speculative writes eagerly, and others lazily. If the write is to be done eagerly, it will be done in the conventional manner. The processor first checks for write permission, 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.

If the write is to be done lazily, the processor initially checks for *read* permission only, not write permission. If the read permission is not available, it requests it by sending a `GetS` request. This ensures the transaction will be notified of any later writes to the line; in that case, it will abort. However, the write permission is not acquired yet.

Once a core has read permission for a line, the address of the line is added to a set, the `LazySet`, and the write is performed at the local private cache. The line is now dirty in the private cache, but invisible to other threads. Additional speculative reads and writes to the line by the transaction will retrieve or store data from/into the private cache, without triggering any coherence actions.

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 a `GetM` request). In other words, the lazy write is now *converted* to an eager one. Any other transactions that have speculatively accessed the line are aborted (by receiving the `GetM` message). From the committing transaction's point of view, although the write into the cache and the obtaining of write permission were done in a different order, the end result is still the same: the core speculatively updated data with the correct write permissions.

Delaying obtaining write permission for a line until commit time makes the conflict detection lazy. Other transactions would previously have seen the request for permission earlier, i.e., eagerly, and now see it at commit time instead. Otherwise, other transactions are not affected.

The second phase is the *Commit* phase. This phase reuses the baseline best-effort HTM commit mechanism, by clearing the transactional bit from all speculative data, making all speculatively written data visible to other threads.

Note that Forgive-TM does not require commit arbitration like some other lazy HTM proposals do. In those proposals, once a transaction starts the commit phase, the phase must not be interrupted. If two committing transactions were to have a data conflict, this would lead to an atomicity violation. Therefore, a special message is sent to its peers to request for commit permission, to hold off other transactions from committing.

In comparison, Forgive-TM simply reuses the existing mechanisms of best-effort HTMs. If a transaction in the *Commit-Prep* phase gets notified of a data conflict, the transaction can simply abort since the transaction is not yet complete. If the transaction in the *Commit* phase gets notified, the core completes the transactional bit clearing and then provides the newly updated data to the requester.

Hardware may choose to convert a lazily written line to an eager one at any time before the transaction is committed. This means that a write can be done lazily at first, but later released as an eager write, e.g. to make room for another lazy write; if we keep a finite-size `LazySet`, we may do this on a `LazySet` *eviction*.

While Forgive-TM enables some writes to be handled lazily, it very intentionally treats most eagerly, for two reasons. First, not all writes benefit from being handled lazily [9], e.g., writes to private data. Second, lazily done writes can impact commit

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

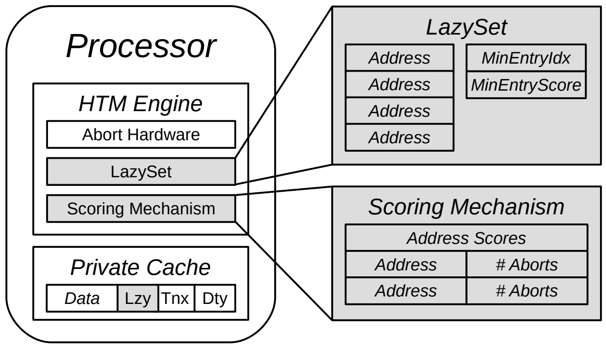


Fig. 3: Architecture

latency [6]. These writes do not need to be done lazily, while others will see benefit. In addition, we may only have room for a small `LazySet`, and need to prioritize which lines should be in that set. Forgive-TM includes a scoring mechanism that determines which writes to do eagerly and which to do lazily (details explained later in Section III-C).

A. Hardware Overview

Figure 3 shows the hardware overview of our proposal. The items in white are common with 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 a speculatively written line has both `Tnx` bit and `Dirty` bits set.

The items in gray are new. First, each cache line is augmented with a `Lazy` bit. This bit indicates whether the line has been written to lazily or not.

Second is the `LazySet`, which contains a set of addresses that were written to lazily. It also contains the score of the address. At the start of the transaction, this set is empty. As the transaction issues writes, the set fills up. Each time a new entry is added to the 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.

Next is the `Scoring Mechanism`. Because the `LazySet` is of limited size, each cache line has a score associated with it, to arbitrate which lines should be in the `LazySet`. The score attempts to approximate the probability that the line will be involved in a data conflict. In Forgive-TM, we use the number of aborts caused by a data address as the score. We discuss how scoring is done in detail, along with possible alternatives.

B. LazySet Maintenance

The `LazySet` is a structure that stores a set of addresses (of cache lines) which were lazily written to. Tracking this set serves two purposes. First, from it we can determine whether an address was previously written in a lazy manner. Second, using it, we can quickly determine which addresses the *Commit-Prep* phase needs to handle.

During transaction execution, every time a write operation is about to be done, the `Lazy` bit is checked. If set, the address was already written to lazily, and the cache line is already part of the speculative memory state (the private L1 cache in our case). All subsequent writes to this cache line are done lazily.

If not set, but the line's `Dirty` and `Tnx` bits are set, the line was written eagerly, and subsequent writes to this line are done eagerly as well.

In both of the above cases, the transaction had already written the line. However, the line might not be in the private cache, or might have only been read so far. In this case, we need to check if the `LazySet` has space. As long as the `LazySet` has space available, all writes are done lazily. The destination address of the operation is added to the set, and the score of the operation is computed using the `Scoring Mechanism`, and updates the `MinEntryIdx` and `MinEntryScore` if needed.

Once the `LazySet` is full, any newly written lines must compete for space. 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 replaced with the new address. The old entry is *converted* into an eager write, by sending a `GetM` request for the address. The new write is done lazily.

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 not added to the LazySet, and is instead done eagerly.

Although both the private cache and the `LazySet` can experience evictions, the end result is different. If a cache line is 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 lines written to eagerly or lazily.

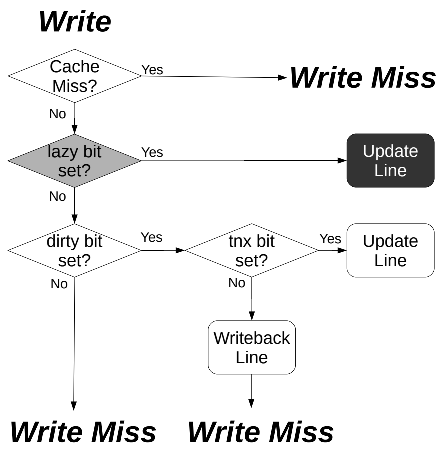
C. Scoring Mechanism

Writes exhibit several different behaviors [9]. Some writes are for managing the call stack, for pushing and popping function local variables. Other writes are to private data. Finally, some writes are to shared data, and it is these writes that an HTM system should treat specially.

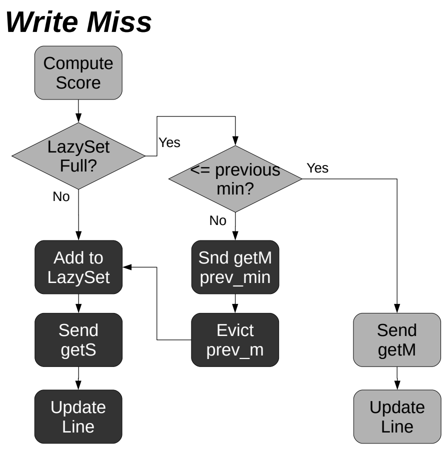
In addition, specifically for HTMs that detect and handle conflicts lazily, the number of writes can directly impact the commit latency. Each lazy write results in more operations that need to be handled during the commit phase [6], [9].

As a result, Forgive-TM limits the size of the lazy set and includes a mechanism to rank cache lines and do lazy writes on only a subset of them. Each cache line that is written is given a score. The score is a proxy of the likelihood that doing the write eagerly will result in data conflicts. Lazy writes may still trigger data conflicts, but doing writes lazily reduces the window of vulnerability, as shown in Figure 1.

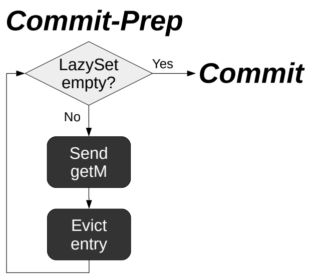
In Forgive-TM, we use the number of aborts triggered by that address as the score. Each time the local transaction is aborted,



(a) Writes



(b) Write Misses



(c) Commit-Prep

Fig. 4: Operation Flowcharts

the counter for the abort-triggering address is incremented. We note that our scores are only "hints", i.e. a "wrong" score reduces the chances of avoiding an abort, but does not impact correctness. This allows us to use a limited-size table.

We note there are other ways to compute scores of these operations, and Section IV examines some of the alternatives.

D. Operation Flowcharts

In this section, we describe in detail how certain operations are handled. First, Figure 4a depicts a flowchart of how Forgive-TM handles writes. 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 4b.

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, we check to see if we have write permission. If we do not, 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 data contents of this line are *not* speculative, and this is the only copy of the data in the system. Therefore, the data needs to be written back before the write miss procedure is handled.

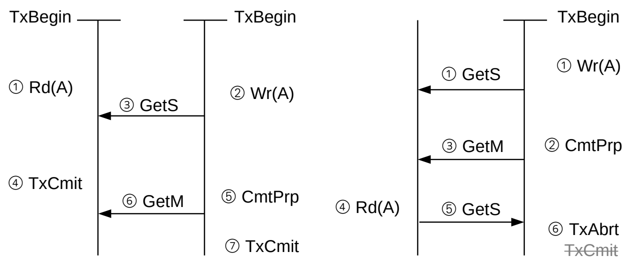
If the transactional bit is also set, then we simply update the line. This case is where the line was already eagerly written, so we just write it again.

Figure 4b is a flowchart of how write misses are handled 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 hardware adds 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, the previous min is *converted* into an eager write, by requesting write permission via a `getM` message. 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`, the write operation is done as usual, by sending a `getM` request and updating the cache line contents.

Figure 4c shows what is done in the `Commit-Prep` 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. This happens while still inside the transaction, before the official commit phase. Once all of the addresses are properly converted into eager writes, the regular commit starts.



(a) Writing Lazily (b) Transaction Abort

Fig. 5: Examples in Forgive-TM

E. Examples

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

First, Figure 5a depicts an example of how writes can be done lazily. As in Figure 2, we have one transaction that does a read (①) and one that does a write (②). This time however, the core issues a `GetS` request (③) 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 the `LazySet`.

Later, the transaction on the left hand side is ready to commit (④). Because from the transaction's point of view, there was no write to address A, the transaction is free to commit. In effect, the transaction has "happened before" the right-hand transaction.

The right-hand transaction is then ready to commit, and the TM hardware starts the Commit-Prep phase (⑤). The TM hardware steps through each entry in the lazy set and issues `GetM` requests to each (⑥).

Once each line is properly accounted for, the TM hardware starts the regular commit operations (⑦). From the thread's point of view, the transaction is complete, with read and write permissions as needed for all speculatively accessed lines. 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.

Figure 5b shows an example of 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), where the HTM hardware steps through each entry in the lazy set and requests write permission for each line (③). However before the core acquires write permission for all lines, another core requests 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).

F. Discussion

Transactional store operations have a *window of vulnerability*, during which another thread can trigger a data conflict abort. As shown in Figure 1, eager conflict detection exposes a larger

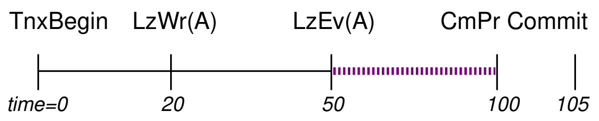


Fig. 6: Overflowed Lazy Write

window of vulnerability compared to lazy conflict detection. By doing many writes lazily, Forgive-TM reduces the window of vulnerability for many of these store operations, improving program scalability.

However, not all stores are exclusively eager or lazy. Forgive-TM may do some writes initially lazily, but later convert them into an eager write. This happens because the lazy set has a limited size. Even for these types of writes, the window of vulnerability will be smaller than if it were written to eagerly as soon as it occurred.

This scenario is depicted in Figure 6. Although the write was initially done lazily, at cycle 20, the address was evicted from the lazy set at cycle 50. In this case, the window of vulnerability is 55 cycles, from the eviction from the lazy set, until the commit. This is smaller compared to the example in Figure 1a, which has a vulnerability window of 80 cycles. Although 55 cycles is larger than a fully lazy write's window of 5 cycles, as in Figure 1b, this is still an improvement. We will look at how the window of vulnerability changes later in Section IV.

Forgive-TM builds on a baseline HTM system that is similar to Intel's TSX [1]. Transactional reads load data into the L1 and L2 cache, and mark the L1 block as transactional. Transactional eager writes takes advantage of the writeback nature of the L1 cache and update the L1 cache only. Forgive-TM builds on this by implementing lazy writes by updating the private, transactional storage without acquiring write permissions (only read permissions). The cache that contains the original data is left alone, for both eager and lazy writes. Although Forgive-TM does not require a writeback L1, it should not be difficult to implement Forgive-TM on a writethrough-based HTM system (e.g., Power [3] or BlueGene/Q [30]).

The baseline TSX implementation provides strong isolation [31]. Forgive-TM provides the same strong isolation, and non-transactional operations will view the entire transaction as a single unit. However, as discussed by Dalessandro and Scott [32], guarantees provided by transactional memory are orthogonal to the memory consistency model, and as such strong isolation does not provide transactional sequential consistency.

IV. Evaluation and Results

To evaluate the effectiveness of Forgive-TM, we modified an architecture simulator to support hardware transactions similar to Intel's TSX [33], [34]. The simulated architecture is an 8 core, 4-way out-of-order processor with private L1 and L2 caches. The 32KB L1 cache acts as a private cache for speculative data, while the 128KB L2 cache stores the original data. The cores are connected to each other using a shared bus, and the

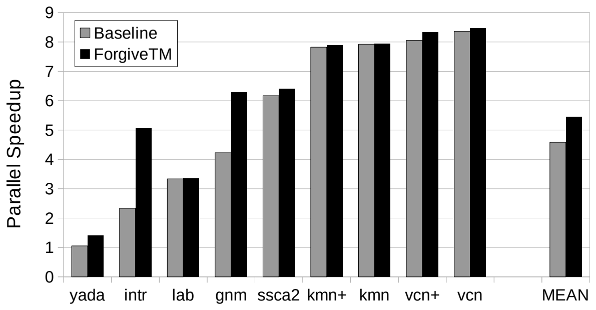


Fig. 7: Overall Results

L3 cache (8MB) is shared. The scoring table is 64 entries, and the lazy set is 16 entries.

For applications, we use the STAMP benchmarks, with updates to more closely match the draft C++ transactional memory support [35], [36]. Bayes was excluded since the runtime is not deterministic. Heap allocation calls were converted to use per-thread pools to reduce data conflicts on these items. Since HTM does not ensure forward progress, each thread keeps a private counter of the number of times it's attempted a transaction. On conflict aborts we retry and increment the counter. However, if the counter is already too large (we use a threshold of 12 attempts), we fall back to turning a transaction into a critical section by acquiring a global mutex lock (which aborts outstanding transactions and prevents any from starting) [37]. All experiments were run with recommended simulator input.

A. Analysis

The overall performance improvement can be seen in Figure 7. Forgive-TM achieves 33% performance improvement in yada, 49% in genome, and over 2.1x improvement in intruder. There is limited improvement with kmeans, labyrinth, ssca2, and vacation since there is a limited number of conflict aborts in these benchmarks in the first place. Overall, Forgive-TM gives a geometric mean speedup of 19% across the benchmarks.

It appears that we achieve superlinear speedup with vacation, even with the baseline sytem. In vacation, the threads manipulate a shared red-black tree. Depending on the order of operations, this can lead to a different tree organization and different tree height. If aborts are infrequent enough (as shown here), we sometimes observe performance greater than the number of threads would seem to allow.

Figure 8 looks at the distribution of the types of aborts. *Eager/Eager* aborts are data conflict aborts between eagerly accessed (and detected) read and write operations. *Eager/Lazy* aborts are between an eager access and a lazy access. This most often happens when a lazy write conflicts with an eager write (because the lazy write acts like an eager read), or vice-versa. *Commit* aborts occur when a transaction with lazy writes is in the *Commit-Prep* phase and converts a lazy write to an eager one; this may trigger a conflict with eager or lazy accesses in concurrent transactions. The baseline scheme will has no *Eager/Lazy* or *Commit* aborts. *Fallback* aborts occur when the fallback path is 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).

We show the number of aborts of each type, normalized to the total number of aborts for the baseline HTM. Lazy conflict detection can significantly reduce the number of aborts, in some cases up to 1/5, and by an average of 39%. Most of the remaining data conflict aborts have been converted to lazy ones, triggered at pre-commit time. In all but genome and yada, *all* of the eager conflict aborts were converted to *Commit* aborts.

Figure 9 shows how the window of vulnerability is affected by introducing lazy writes. Recall that each speculative write results in a window of increased vulnerability for the transaction, because the additional entry in the transaction's write set can cause the transaction to be aborted if any other thread reads (or writes) the line. We quantify the window of vulnerability for speculative stores by normalizing its length to the length of the transaction. Thus, a 1.0 indicates that the duration of the entire transaction is the line's window of vulnerability, whereas 0.0 indicates that there is no window.

We compute a weighted average of all written lines to get the overall window of vulnerability for a given transaction. The weights are computed using the number of times the address was responsible for an abort. The idea is that the more "troublesome" a data address is, the more important it is to consider its window of vulnerability. [2]

We then compute a weighted average of all successful transactions' windows to get a single vulnerability metric for each run. The weights in this case are the duration (in cycles) of each transaction. The rationale here is that the longer the transaction is, the more important it is to have a small window of vulnerability.

For all benchmarks, Forgive-TM greatly reduces the window of vulnerability. On average, the window is 77% smaller for Forgive-TM than for the baseline HTM (1/7). This helps explain the large drop in aborts in most benchmarks in Figure 7.

One concern with lazy writes is that waiting until commit time to do the write operations can extend the length of the commit operation itself [9]. Because the coherence operations are delayed, this means we have less work to hide the latency of that operation.

We examine this overhead in Figure 10. We compute the overhead of the `Commit-Prep` phase by computing the ratio of time in `Commit-Prep` versus the transaction body. In most cases the overhead of `Commit-Prep` 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

[2]We considered using the maximum window across all lines for each transaction as that transaction's window; however, we found that misleading since it overestimates the importance of "uninteresting" stores, such as those to the stack.

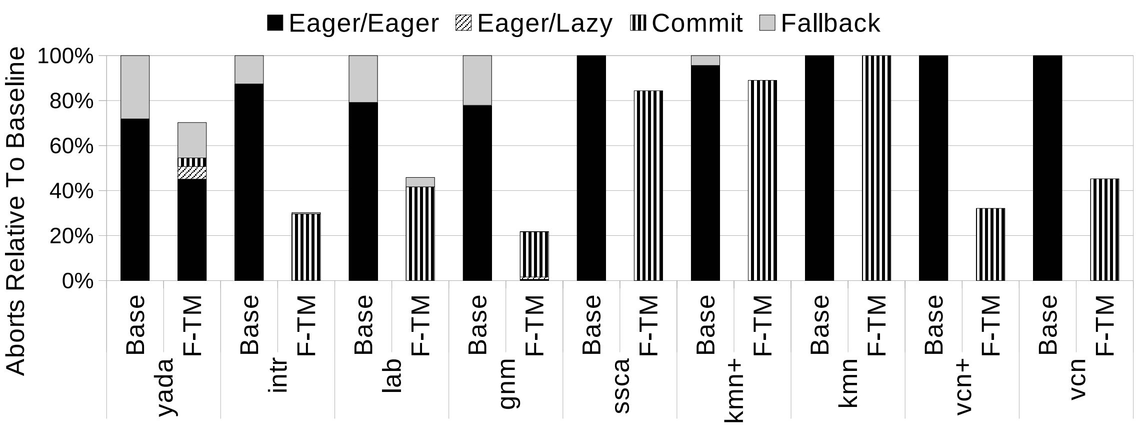


Fig. 8: Abort Types

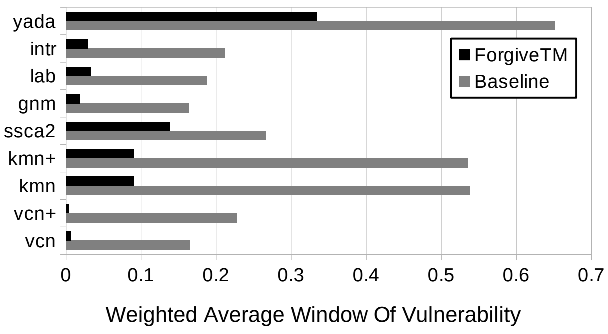


Fig. 9: Overall Window of Vulnerability

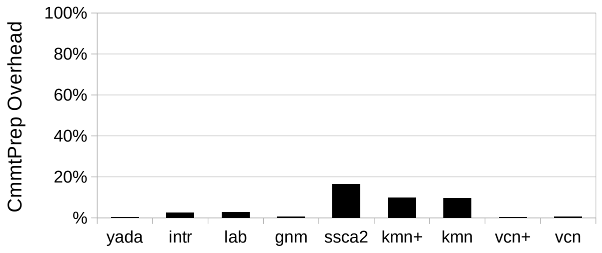


Fig. 10: Overhead of Commit-Prep

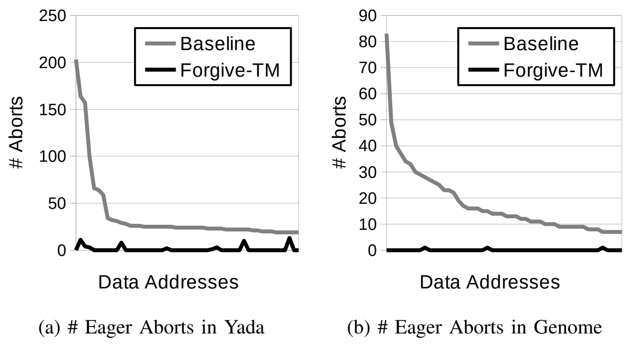


Fig. 11: # Eager Aborts Triggered by Each Address

parallel. A side effect of our mechanism is that a transaction may encounter a data conflict during the *Commit-Prep* phase and be aborted. However, this was rare in our experiments.

In Figure 11, we investigate in more detail how lazy conflict detection improves performance over the baseline configuration. We look at the histogram of the number of conflict aborts due to eager writes for two benchmarks. We sort all lines according to their conflict aborts in the baseline HTM and show data for the top 50.

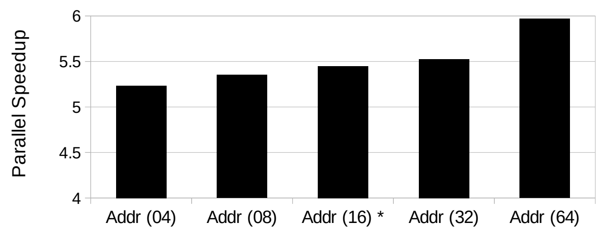
There are two things that can be noticed here. First, under the baseline configuration, there is significant skew in the number of aborts each address causes. In comparison, under Forgive-TM, the distribution of such aborts is much flatter. While some aborts are transformed from eager to lazy ones, as can be seen in Figure 8, 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.

B. *Scoring Mechanism Sensitivity Studies*

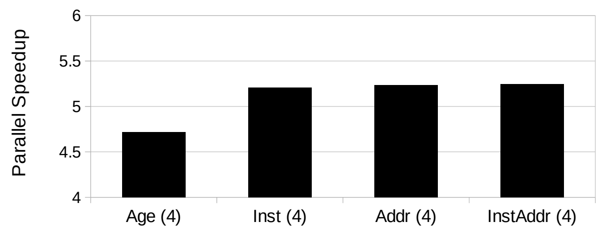
We now perform sensitivity studies on two aspects of the lazy set: the size of the lazy set, and the scoring mechanism. Not all writes are equal, and some writes are more “conflict-prone” than others [9]. Therefore, Forgive-TM computes a score for each write and only does writes lazily to those that are most likely to lead to a future conflict.

First, in Figure 12a, we look at how the size of the lazy set affects overall performance. The larger the lazy set, the more writes that can be done lazily. The scoring policy used here is the *Addr* policy, which will be explained later.

We consistently see higher performance with a larger maximum lazy set size. Although a large lazy set can result in

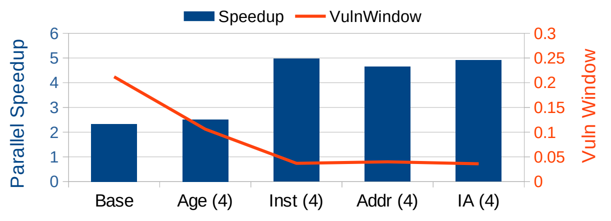


(a) Using Different LazySet Sizes

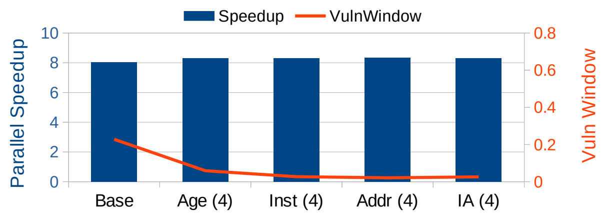


(b) Using Different Scoring Policies

Fig. 12: LazySet Management and Performance



(a) Intruder



(b) Vacation

Fig. 13: Per-policy Window of Vulnerability Trends

a longer *Commit-Prep* phase overhead, the aborts avoided by doing lazy writes can more than overcome the *Commit-Prep* overhead.

In Figure 12b we compare several scoring policies. Once the lazy set is full, there needs to be some mechanism to discriminate writes. For this experiment, we use a 4-entry lazy set, to emphasize the difference between policies.

The *Age* policy prefers older writes over younger writes within a transaction. Under this policy, hardware adds all writes to the lazy set until the set is full, and never evicts anything from the lazy set. 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 instructions involved in many data conflict aborts are likely to trigger them again. When the first speculative write for a transaction happens for a given line, the PC of the write instruction is saved in an *instruction score table*, along with the address that was written. Later, when a coherence request results in a data conflict abort, the address is matched in the instruction score table, and the corresponding PC's score is incremented.

We separate store instructions into just two classes: those that have been involved in aborts and those that have not. We tried schemes that kept a continuum of scores, but they worked no better. The key benefit comes from differentiating "troublesome" stores from other stores.

The *Addr* is the default Forgive-TM policy, described in Section III-C and evaluated earlier. Each core individually keeps track of the number of aborts triggered by each cache line. The idea is that a data address that triggers a lot of aborts, is "hot," and will continue to be popular.

Last, the *InstAddr* policy uses the sum of the scores from *Inst* and *Addr*. This policy tries to take the best of both the instruction-based scoring policy and the address-based scoring policy. When combining scores, the binary score from the Inst policy is treated as either a 0 or a 2.

As can be seen in Figure 12b, scoring policies does have an affect on overall performance. Instead of just issuing all writes lazily until the lazy set is full (*Age*), having some sort of knowledge on the likelihood of future conflicts improves performance by 10%, and up to 1.8x in intruder. Between the three other policies (*Inst*, *Addr*, *InstAddr*), the performance results are similar, although for some policies work better for some benchmarks than others. Because maintaining instruction-based scores requires additional complexity since we need to track which instruction triggered the first write to each cache line, to assign blame to an instruction on an abort, Forgive-TM implements the simpler policy, the *Addr* policy.

We drill a bit deeper into the impact of the scoring policy. Figure 13 looks at the average window of vulnerability for two selected benchmarks, intruder and vacation, under different scoring policies. Recall from Figure 9 the window of vulnerability for a given benchmark is computed by taking the (weighted) average window of each transaction, which itself is the weighted average of each write operations' window. We show the parallel speedup and weighted average window of vulnerability for baseline, and Forgive-TM using *Age*, *Inst*, *Addr*, *InstAddr* (IA) policies.

The baseline policy has the largest window of vulnerability. Under Forgive-TM, a subset of the write operations are converted into lazy writes, which reduces the size of the window. Since fewer write addresses are exposed to potential conflict, the number of conflict aborts tend to decrease, improving performance.

For intruder, the baseline has a large window of vulnerability, making it quite sensitive to data conflict aborts. Although converting eager writes to lazy writes will reduce the window, the size of the lazy set is limited. Instead of wasting space in the lazy set, it becomes important to be more selective. As the policy makes better decisions, the performance increases.

In comparison, under vacation there is almost no performance sensitivity to the treatment of stores. The window of vulnerability is small even for the baseline, and the parallel speedup for the baseline is already excellent. Thus, while Forgive-TM is doing what it's intended to do, shrinking the window of vulnerability, there is little room for performance improvement.

V. Related Work

TCC [8] was 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 a 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 [26] and BulkSC [27] take a similar approach to TCC, but overcome the requirement of a shared bus by adding a commit arbiter. ScalableBulk [28] and BulkCommit [29] 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 [5] adds transaction abort request messages to the coherence protocol instead. While executing a transaction, a core sends special TM specific requests to notify other cores that its transaction is currently working on a given address. This allows transactions to build a set of conflicting transactions so, when a transaction is ready to commit, it can send `abort` requests to all transactions that it knows have performed conflicting memory accesses.

PI-TM [12] 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.

LV* [13] also tracks sets of conflicting transactions at each core. During execution, transactions snoop coherence messages to keep track of with which other transactions 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 clear its killer map entry in the other cores.

SI-TM [38] is a more ambitious proposal that provides *snapshot isolation* in hardware TM, instead of the usual *opacity* guarantee that other systems provide [24]. Snapshot isolation is a weaker correctness guarantee 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.

Like Forgive-TM, there have also been proposals to take an existing HTM and add lazy conflict detection capabilities. However, these proposals use LogTM as their baseline HTM. Transactional Store Buffers [10], [11], similar to Forgive-TM, do not immediately do a coherence state change when issuing a lazy write. However, at commit time, transactions start commit arbitration to make sure the winning transaction can commit exclusively. In addition, taking advantage of the fact that LogTM can provide sophisticated conflict resolution schemes, cores with the commit token will have their coherence messages flagged to always win conflicts with other sharers (i.e. committers NACK other cores).

SEL-TM [9] is similar in that it builds on top of LogTM. However, instead of a first-come-first-serve scheme (using an *Age* based scheme in our terminology), SEL-TM uses a sophisticated scoring mechanism to prioritize conflict prone addresses. However, the scoring mechanism is guided in part by the ability to NACK conflicting requests and survive them, which do not exist in current best-effort HTMs.

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

FlexTM [39] 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 other transactions at commit time.

DynTM [40] 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. Hardware may instead choose to run a transaction eagerly, where it automatically aborts or stalls conflicting transactions.

Speculation-Based Conflict Resolution [41] also detects conflicts eagerly but handles them lazily. Unlike other schemes, however, transactions with write-after-read conflicts are ordered. When conflicting transactions attempt to commit, the commit is done in the write order, and aborts are avoided.

VI. Conclusion

Conventional hardware transactional memory systems eagerly detect conflicts between transactions by taking advantage of the existing coherence protocol. Coherence requests are used as notifications for possible conflicts with the transaction's read and write sets.

Lazy conflict detection can provide better scalability, but previously proposed implementations require changes to parts of the processor that commercial vendors are reluctant to touch. Forgive-TM allows conventional HTM systems to use lazy conflict detection, while still keeping the coherence protocol unmodified. It does so by delaying the coherence requests for write permission. When the transaction reaches the commit instruction, the HTM system now sends the write permission request messages. This simple reordering of operations allows Forgive-TM to provide better scalability for multithreaded applications.

References

[1] Intel Corporation, "Intel® architecture instruction set extensions programming reference," 2012.

[2] 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, Dec 2012, pp. 25–36.

[3] 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, 2013, pp. 225–236.

[4] N. Stephens, "New technologies in the arm architecture," https://connect.linaro.org/resources/bkk19/new-technologies-in-the-arm-architecture/; accessed 15-Mar-2019.

[5] 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, NY, USA: ACM, 2009, pp. 145–155. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669132

[6] 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. [Online]. Available: http://doi.acm.org/10.1145/1273440.1250674

[7] 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)*, March 2016, pp. 285–296.

[8] 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 *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 102–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006711

[9] 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*, May 2012, pp. 95–106.

[10] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. García, and P. Stenstrom, "Eager beats lazy: Improving store management in eager hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 11, pp. 2192–2201, Nov 2013.

[11] A. Negi, R. Titos-Gil, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Eager meets lazy: The impact of write-buffering on hardware transactional memory," in *2011 International Conference on Parallel Processing*, Sept 2011, pp. 73–82.

[12] ——, "Pi-tm: Pessimistic invalidation for scalable lazy hardware transactional memory," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012, pp. 1–12.

[13] 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. New York, NY, USA: ACM, 2010, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/1882453.1882460

[14] 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)*, Feb 2013, pp. 566–577.

[15] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: verifying memory coherence in the cray x1," in *Proceedings International Parallel and Distributed Processing Symposium*, April 2003, pp. 10 pp.–.

[16] 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. [Online]. Available: https://doi.org/10.2200/S00346ED1V01Y201104CAC016

[17] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *2008 IEEE International Conference on Computer Design*, Oct 2008, pp. 348–355.

[18] "MOESI CMP directory," http://gem5.org/MOESI_CMP_directory.

[19] 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.

[20] O. Matthews and D. J. Sorin, "Architecting hierarchical coherence protocols for push-button parametric verification," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 477–489. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123971

[21] I. Wagner and V. Bertacco, "Mcjammer: Adaptive verification for multi-core designs," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. New York, NY, USA: ACM, 2008, pp. 670–675. [Online]. Available: http://doi.acm.org/10.1145/1403375.1403539

[22] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–251.

[23] M. Zhang, J. D. Bingham, J. Erickson, and D. J. Sorin, "Pvcoherence: Designing flat coherence protocols for scalable verification," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 392–403.

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

[25] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *Intl. Symp. on Computer Architecture*, ser. ISCA '06, 2006, pp. 227–238.

[26] 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*, Feb 2007, pp. 97–108.

[27] 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. New York, NY, USA: ACM, 2007, pp. 278–289. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250697

[28] 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. [Online]. Available: https://doi.org/10.1109/MICRO.2010.29

[29] 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. New York, NY, USA: ACM, 2013, pp. 371–382. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540740

[30] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, and B. Steinmacher-Burow, "Ibm blue gene/q memory subsystem with speculative execution and transactional memory," *IBM Journal of Research and Development*, vol. 57, no. 1/2, pp. 7:1–7:12, Jan 2013.

[31] N. Chong, T. Sorensen, and J. Wickerson, "The semantics of transactions and weak memory in x86, power, arm, and c++," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 211–225. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192373

[32] L. Dalessandro and M. L. Scott, "Strong isolation is a weak idea," in *TRANSACT: 4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[33] 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.

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

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

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

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

[38] 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. New York, NY, USA: ACM, 2014, pp. 383–398. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541952

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

[40] 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. [Online]. Available: https://doi.org/10.1109/MICRO.2010.23

[41] 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*, May 2009, pp. 1–12.