Eager meets Lazy: the Impact of Write-Buffering on Hardware Transactional Memory

Anurag Negi[⋆] Rubén Titos-Gil[†] Manuel E. Acacio[†] José M. García[†] Per Stenstrom[⋆]
Chalmers University of Technology[⋆]
Universidad de Murcia[†]
{negi,per.stenstrom}@chalmers.se
{rtitos,meacacio,jmgarcia}@ditec.um.es

***Abstract*—Hardware transactional memory (HTM) systems have been studied extensively along the dimensions of speculative versioning and contention management policies. The relative performance of several designs policies has been discussed at length in prior work within the framework of scalable chip-multiprocessing systems. Yet, the impact of simple structural optimizations like write-buffering has not been investigated and performance deviations due to the presence or absence of these optimizations remains unclear. This lack of insight into the effective use and impact of these interfacial structures between the processor core and the coherent memory hierarchy forms the crux of the problem we study in this paper. Through detailed modeling of various write-buffering configurations we show that they play a major role in determining the overall performance of a practical HTM system. Our study of both eager and lazy conflict resolution mechanisms in a scalable parallel architecture notes a remarkable convergence of the performance of these two diametrically opposite design points when write buffers are introduced and used well to support the common case. Mitigation of redundant actions, fewer invalidations on abort, latency-hiding and prefetch effects contribute towards reducing execution times for transactions. Shorter transaction durations also imply a lower contention probability, thereby amplifying gains even further. The insights, related to the interplay between buffering mechanisms, system policies and workload characteristics, contained in this paper clearly distinguish gains in performance to be had from write-buffering from those that can be ascribed to HTM policy. We believe that this information would facilitate sound design decisions when incorporating HTMs into parallel architectures.**

I. Introduction

Transactional memory (TM) envisages programming constructs for optimistic concurrency control in parallel shared memory applications. Sections of code that perform such accesses, demarcated as *atomic blocks* or *transactions*, are guaranteed the properties of atomicity and isolation by the underlying system. Considerable book-keeping, necessary to correctly implement TM, makes software implementations cumbersome. Hardware TM (HTM) systems propose architectural extensions to support HTM and leverage existing architectural features like caches and coherence protocols to provide efficient common case performance comparable to fine-grained lock based approaches.

Most HTM designs rely upon containment, in some fashion, of updates made in a transaction. Updates can be confined to thread-private structures like private caches, as is done in [3], [8]. Such *versioning* of possible future values of shared data is termed lazy, where the defining characteristic is that exclusive ownership over speculatively targeted locations is acquired only after a transaction's execution is *guaranteed to succeed*. Alternatively, updates can be made in-place, which implies *early acquisition* of written locations, when protocols exist to ensure their isolation and restoration of a consistent state when data-races need resolution. Such a mechanism is termed *eager* and is utilized by designs like [19]. A closely related design choice is that of conflict resolution. Eager versioning of updates necessitates *eager resolution* of conflicts – races must be detected and resolved when an in-place shared memory update is attempted. Lazy versioning, however, also allows conflict resolution to be deferred until a transaction tries to *commit*, i.e. make its updates globally visible. This *lazy resolution* of conflicts requires *aborting* and re-executing transactions that conflict with the committing one.

Tracking transactional state by book-keeping in private caches is a popular HTM design technique [7], [8], [19]. It allows cache coherence protocols to be leveraged for conflict detection at the granularity of cache lines. To do so, however, requires certain protocol actions be taken that incur a cost in terms of latency when they are performed and the possibility of actions to revert their effect in the future. Take for example a typical multicore HTM design where both versioning and conflict resolution are lazy and directory based MESI coherence is employed. The coherence protocol must allow for the existence of multiple uncommitted versions of the same cache line in private caches written by different transactions. The simplest way to do so would be to have the coherence protocol treat all such transactions as sharers of the last consistent state of the cache line till one tries to commit. This necessitates downgrade of dirty or exclusively owned lines that are targeted by transactional writes to shared state with a possible write-back of consistent state to shared levels of the cache hierarchy. Standard directory invalidations can now be used for discovery of data races. In scenarios where the probability that a transaction will commit without aborting is high, such coherence actions to enable conflict detection prove to be redundant in the common case. It is quite likely that, due to locality of reference and low contention, transactional updates hit lines that are already dirty or exclusively owned by the private cache. Downgrading these lines to shared state to correctly handle the unlikely case when a conflict might be seen on the lines is inherently pessimistic and penalizes the common, non-

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.63

IEEE computer society

contended case. This penalty can be circumvented by buffering transactional updates emitted by the processor before they reach the coherent private cache.

Most modern microprocessors use such write buffers to hide latency for completing updates to memory. Combined with a store-forwarding mechanism, the technique quite effectively avoids stalling when write-misses are encountered. Such buffers when suitably modified for containment of transactional updates can provide significant reduction in HTM protocol overheads. In lazy designs, updates in the buffer would be released to the coherent cache hierarchy only when a transaction attempts to commit or when the buffer overflows. If, in the common case, most transactional updates can be contained in this structure, redundant protocol actions would be significantly reduced. It should be noted that in multilevel private cache hierarchies the concept of write buffers can be extended to private L1 caches that need not maintain coherence on transactionally updated lines until commit.

An analogous situation arises when coarse grained transactions show relatively high contention on a relatively small portion of updated lines. In such cases a large number of aborts would occur resulting in invalidations of uncommitted lines in the private cache, a large fraction of which do not exhibit contention. When the transaction re-executes, it will encounter costly private cache misses when such data is accessed again. Buffering of transactional updates before they reach the private cache will largely eliminate such misses in the common case. On aborts, only the contents of the buffer would be discarded. Confinement of writes in the buffer till commit also avoids writer-writer conflicts on such locations since they do not contaminate cache lines and hence result in no ambiguities when the final cache line update occurs.

Write buffers show interesting effects when HTM designs employ eager conflict resolution. Typically, eager HTMs detect conflicts when a write from the core is received by the cache hierarchy. The core waits for the result and takes corrective action if a conflict is reported. With write-buffering, the processor is decoupled from conflict detection mechanisms and can continue execution past a potentially conflicting update, a feature typically found in lazy HTM designs. Buffered writes are released in a controlled manner into the private cache. An in-place update in shared memory is attempted in a non-blocking manner while the processor continues to run ahead. Logging of old values in the undo-log as necessitated by eager versioning is also taken out of the critical path. In case a conflict exists, the request can be retried in the background while execution of the transaction continues. Thus, transactional updates never stall execution on the core.

In this paper we demonstrate that in both eager and lazy conflict resolution schemes, write-buffering can achieve substantial reductions in transactional execution times due to the interplay of conditions described above. This, in turn, results in a contraction of the window of contention for concurrent transactions amplifying performance gains even further. While the utility and ubiquity of write buffers in standard microprocessors is well recognized, their use and implications in the context of HTM have not been studied in depth in prior work. This paper attempts to fill this gap and shows how these structures impact transactional behavior of a diverse set of workloads. It quantifies performance gains and reductions in redundant work and stall times achieved by write buffering for several pertinent design points. The paper also highlights a remarkable leveling out of average performance metrics for eager and lazy systems, showing that with well-balanced write buffers of modest sizes eager systems are as good as or better than lazy designs. Prior research in this area shows a bias favoring lazy HTM designs, particularly when high contention workloads are concerned. Another contribution of equal significance is the underlining of the importance of modeling standard processor structures accurately after applying straightforward optimizations to such structures, in order to support transactional memory more effectively. Careful identification of sources of performance variation is necessary before ascribing it to changes in policy. Not doing so can result in measurements that are substantially different from what might be seen in a real-world implementation and, more importantly, can lead to erroneous biases in favor of or against certain design options.

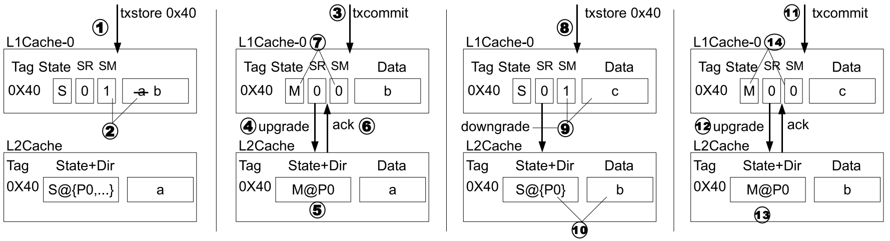
II. Buffering speculative updates in coherent caches

This section discusses how speculative updates are buffered in coherent caches in both lazy and eager designs. Lazy designs need discussion since they require modest deviations from the way coherence protocols typically work. Eager designs do not require behavioural changes in the protocols to support transactional updates but such updates have performance implications, nevertheless.

We have chosen a tiled chip-multiprocessor (CMP) design as reference because its modular nature makes it popular in several commercial many-core designs and the availability of versatile simulation setups [12] makes the modeling and comparison of policies and architectural features less daunting. The basic architecture comprises several tiles overlaid over a point-to-point interconnect forming a mesh-based network-on-chip. Each tile has a processing core, one level of private cache, a slice each of the shared inclusive level 2 cache and the corresponding directory entries and some routing logic. A MESI protocol is used to keep private caches coherent.

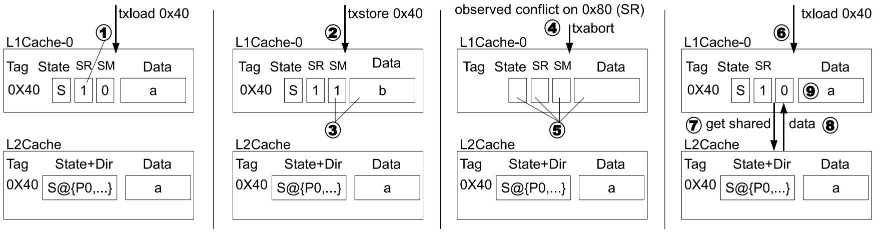
A. Lazy HTMs

To buffer speculative data in private caches, per-cache line meta-data is augmented with two bits, SR and SM, which indicate whether a line has been speculatively read or speculatively modified, respectively. During the course of execution of a transaction writes appear as non-invalidating reads to the coherence protocol. In order to preserve its last globally consistent value, a dirty (non-speculative) line is written back to the shared memory hierarchy prior to the first speculative update to it in a transaction, resulting in a downgrade of its coherence state from M to S. Commits imply acquisition



1-2. Store writes spec value b to L1 and sets SM bit. **3-4**. Transaction commits and requests coherence upgrades for all SM lines to L2. **5-6**. L2 directory invalidates all other sharers and acknowledges upgrade, P0 becomes exclusive owner for line 0x40. **7**. SM lines in L1 transits from shared (S) to modified (M) state, and SM bit is cleared. **8-10**. New transaction begins; another store to same addr is executed. Before the speculative value 'c' is written to L1, the last committed value 'b' is written back to the L2 and the coherence state is downgrade from M to S. **11-14**. Transaction commits and L1 needs to re-upgrade its coherence state for the line again (downgrade miss)

Fig. 1. *Downgrade miss*: Redundant cache-state changes when a transaction eventually commits.



1. Load hits in L1 and sets SR bit. **2-3**. Store writes spec. value b in L1 and sets SM bit. **4-5**. A conflict on other SR line causes the abort and consequent invalidation of all SM lines. **6-9**. Transaction restarts and this time the load misses in L1 (abort miss)

Fig. 2. *Abort miss*: Invalidations that could be avoided using a write buffer.

of ownership over all lines with SM set, while aborts imply invalidation of all such lines.

Let us now consider the case presented in Fig. 1. A line that is only written by one transaction (it is either thread-private or not actively shared in the current phase of the workload) might, in the steady state, be found with high probability in M state in the private cache. To preserve the old content of the line in the absence of non-coherent write buffering, it must be written back prior to the write, resulting in a transition to S, as shown in Fig. 1 (step 9). On commit we need to reacquire exclusive ownership to the line. Since such a line will not have any sharers, this work (M⟶S and S⟶M) to ensure no races exist is largely redundant. We refer to such events where unnecessary work prolongs the commit phase of a transaction as a *downgrade miss*. Coarse grained transactions that have a relatively large fraction of the write set as private data (stack, thread-local storage) are expected to show the most degradation in performance. In Section IV we examine in detail the impact of downgrade misses and see that for applications like genome and vacation [2] their elimination results in a significant contraction of commit delays.

Fig. 2 depicts the alternate case. A transaction speculatively updates a non-contended line present in its cache and then aborts. As depicted in Fig. 2 (step 6), the line would not be found in the private cache on re-execution as all lines in the write-set have now been invalidated. We refer to such an event as an *abort miss*. Such misses have also been referred to as *contamination* misses [18]. Workloads with large write sets and high contention over small amounts of shared data would experience the greatest drop in private cache hit rates. As Section IV will show, elimination of abort misses using write buffers results in a marked improvement in L1 hit rates.

Some transactional workloads also suffer from write-write conflicts. These are not true conflicts (not data races per se) but need to be resolved in invalidation-based coherence protocols. Consider two transactions that write (but do not read) to a certain cache line. When one commits the other needs to be aborted since the invalidation message only tells us that the cache line might need merging and that cannot be handled without making the coherence protocol a lot more complex.

B. Eager HTMs

Eager HTMs do not require special coherence actions to be taken when updating lines. Yet, they require the old value of the line to be preserved in a special undo-log, a part of which may exist as a hardware structure in the private cache and a part implemented as a data structure in virtual memory. In the architectural setup for this study, the cache has a small 8 cache line *old-value-buffer (OVB)*. Overflows from the OVB are accommodated in the software log.

Before a transactional write can be performed, exclusive ownership permission on the target cache line is required. Thus, for shared or absent cache lines the write must be buffered in the standard miss state handling mechanism, MSHR. If the old-value logging operation misses in the OVB an update to the software log must also be initiated which can result in a second in-flight access being added to the MSHR. The old value of the line must be preserved till the software-log cache line is allocated in the cache. If a race with another transaction causes a coherence request to be retried, the request can sit in the MSHR being sent out periodically into the network. Thus, it can be observed that a number of actions that operate on entire cache lines may be invoked when a transactional store hits the coherent cache hierarchy resulting in increased cache controller usage, traffic and even stalls when stores are bursty.

III. Use of write buffers

Private caches are present primarily to keep frequently used data close to the processor core. Their use in buffering uncommitted data should be made conservatively. Write buffers between the core and the coherent first level cache can be used to prevent transactional updates from polluting the coherent cache hierarchy. The idea can also be extended to inclusive two-level private caching schemes, wherein the first level private cache can be made non-coherent when handling transactional updates.

This can be incorporated very simply into the design by capturing writes issued by the processor and then releasing those to the private cache in a controlled manner. In a lazy HTM, writes would be captured throughout the execution of a transaction or as long as buffer capacity is not exceeded. On commit, the buffer would be flushed causing all writes to enter the memory hierarchy as quickly as possible. On abort, the buffer contents would simply be discarded. In an eager HTM, it would obviously also participate in any store forwarding mechanism. In fact, existing write buffering schemes could be suitably modified to enable such functionality.

It can be observed quite easily that unnecessary switches in coherence state and invalidations of aborts can be completely eliminated if write-sets are fully contained in write buffers. Since speculative data can be recorded in the write buffer, we can eliminate write-backs and downgrades of M lines to shared (S) state. On commit, since the line would likely be present in the cache in the M state, it can simply be written into the private cache without any coherence action. An abort results in the speculative contents inside the write buffer to be discarded. No cache lines need invalidation and, thus, the transaction on re-execution would still find such lines in the private cache. Writes contained in the write buffer can be probed for cache-hits and if they are not found in the cache a non-exclusive prefetch can be made. This is done in the expectation that a long latency write-miss at commit can now be converted to a potentially less costly upgrade-to-exclusive request. Later in this study, we find that doing so improves performance in most cases.

Another benefit of having write buffers is reduction in the number of write-write conflicts that are an artifact of using coherence messages to detect possible modifications to different parts of the same cache line. The cache, when it does not buffer any speculative updates, only records the read set of a transaction. Hence any invalidations resulting from transaction commits result in aborts only when there is a possible true data race, i.e. the write set of the committer conflicts with the read set of the other. We would like to point out that this does not eliminate conflicts due to reader-writer false sharing.

TABLE I
System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split 4-way, 1-cycle latency
Write Buffer	128 bytes
Old Value Buffer	8 cache lines
L2 cache	Shared, 512KB per tile, unified 8-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

Eager HTMs also benefit from the reduction in the number of abort misses. They also benefit from hidden latencies for eager conflict detection on stores. When executing transactions, there are several choices for write buffer behaviour. This relates to how buffered writes are drained into the coherent memory hierarchy. We define *immediate draining* as issuance of the update into the coherent hierarchy as soon as possible after the write has been buffered. This allows early detection of conflicts and hiding of memory fill latencies. *Deferred draining* delays the issue till a later point in the execution of the transaction, based on criteria like the fraction of used entries in the write buffer being greater than a certain threshold. The write buffer is drained on transaction commit. Deferred draining allows greater opportunities for speculation and mitigation of write-write conflicts.

IV. Methodology and Evaluation

A. Experimental Setup

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set [12], in conjunction with Virtutech Simics [11]. We use the detailed timing model for the memory subsystem provided by GEMS, with the Simics in-order processor model. Simics provides functional simulation of the SPARC ISA and boots an unmodified Solaris 10. We perform our experiments on a 16-core tiled CMP system, as described in Table I. The L1 caches maintain inclusion with the L2. The private L1 caches are kept coherent through an on-chip distributed directory (associated with L2 cache banks), which maintains a presence-bit vector of sharers and implements the MESI protocol.

Lazy designs. The lazy HTM system modeled in this evaluation is an extension of the lazy system considered by Bobba et al. [1], available in the GEMS v2.1 release. While Bobba's

TABLE II
WRITE BUFFER CONFIGURATIONS.

Lazy	
infWB	infinite write buffer
noWB	no write buffer
realWB	finite write buffer
realWB_pf	finite write buffer with prefetch
realWB_pf_pc	finite write buffer with prefetch and parallel commit-time write-set acquisition
Eager	
EE_base	GEMS baseline
EE_lw	GEMS baseline with lingering write optimization
EE_infWB	infinite write buffer
EE_realWB_DD	finite write buffer with deferred draining
EE_realWB_ID	finite write buffer with immediate draining

LL system models a private, per processor infinite write buffer, for this study we extended the simulator to precisely model finite buffering for transactional writes. We limited the capacity of the non-coherent write buffer, so that once it fills up, transactional stores happen in the private data cache. Unlike writes to the L1 cache, which need the line present in cache to be able to complete, writes to the write buffer allow the core to execute ahead. A non-blocking *prefetch-read* for the line is sent to the L2 cache if the line is absent in the L1 cache. We modified the replacement policy of the L1 data cache by giving the highest priority to speculatively written lines, in order to minimize the number of transactional overflows when executing large transactions. Nonetheless, we incorporate a speculative victim buffer to avoid serialization penalty due to limited buffering capacity, similarly to [1]. In our simulations only yada experiences a few such evictions and a small victim buffer with 8 entries proves sufficient. We use an ideal book-keeping scheme to track read sets (*perfect signatures*) even when some speculatively read lines have been evicted, in an attempt to isolate our study from the effects of false conflicts arising from non-ideal signature schemes like bloom filters. A simple commit token algorithm is used to serialize transaction commits: Transactions arbitrate for the token using a zero-latency broadcast bus. Once the token is acquired, a transaction enters the commit phase and issues coherence requests to gain exclusive ownership over all lines in its write set. We later present results for the five lazy HTM configurations listed in Table II. Finite write buffers are 128 bytes in size and have word (32bit) granularity. The last two configurations attempt non-exclusive prefetches on cache lines targeted by writes if not already present in the cache. These prefetches do not block execution on the core and are done in the expectation that it will help reduce the quantity of data transferred at commit time. For a fair comparison with eager HTM designs, (Section IV-D), *realWB_pf_pc* speeds up commits by parallelizing issuance of writes into the coherent hierarchy at commit time.

Eager designs. The eager design is based on Log-TM [19]. We have introduced a simple *lingering-write* optimization (referred to as *EE_lw*) in the basic eager implementation. If writes are pending when a transaction aborts they are silenced and completed in background, perhaps even after the transaction has restarted. In fact, the in-flight access (i.e. the lingering write) often proves useful bringing in data required during re-execution. This has two benefits – first, the silenced write acts like an *exclusive prefetch* and second, the transaction can now restart earlier with better chance to use the cache line before it is invalidated due to contention. This significantly boosts performance when contention is high and transactions are small. This simple optimization results in a fairer comparison of policies and effects of other structures. We felt it would be useful for readers to see this compared with the basic protocol available with GEMS and, hence, have included the GEMS reference implementation (*EE_base*) in our results.

The baseline system is then augmented with a finite write buffer. Transactional execution on a core is now not stalled when an update requires coherence actions. It is simply buffered in the write buffer. In contrast to lazy designs, where we attempt to keep writes in the buffer for as long as possible, the eager design drains buffered writes into the coherent hierarchy at some point before commit. We model two configurations – *EE_realWB_ID* which implements immediate draining and *EE_realWB_DD* which implements deferred draining. Immediate draining involves issuing writes into the buffer as soon as possible. Deferred draining attempts to keep writes in the buffer until 80% of the buffer fills up. To implement a limit study, we also model a configuration with an infinite write buffer (*EE_infWB*) that does not drain until a transaction attempts to commit.

Workloads. For this evaluation, we have selected seven (out of eight) transactional applications from the STAMP suite [2]: genome, intruder, kmeans, labyrinth, ssca2, vacation and yada. The application, bayes, was excluded since it exhibits unpredictable behaviour and high variability in its execution time [6], [13]. For kmeans and vacation, both high and low contention configurations were used. Input parameters, detailed in [2], were used. Small input results for all workloads are provided. Where simulation times made it feasible, we have included results for medium length runs – four applications that show widely varying transactional characteristics (ssca2 at the low contention end to intruder at the high contention end) have been included. We expect these results to present a better picture of real-world performance.

B. Lazy HTM Results

In this section, we analyze the impact of the four buffering schemes with lazy conflict resolution described earlier and quantify the effectiveness of write buffers in improving cache performance and reducing the number of coherence actions required on commit. Fig. 3 shows the average miss rate of L1 data caches for each STAMP benchmark. A dedicated write buffer reduces miss rate for almost every benchmark. In Fig. 4, we present the number of abort misses suffered on average by a transaction that restarted at least once. The same plot also

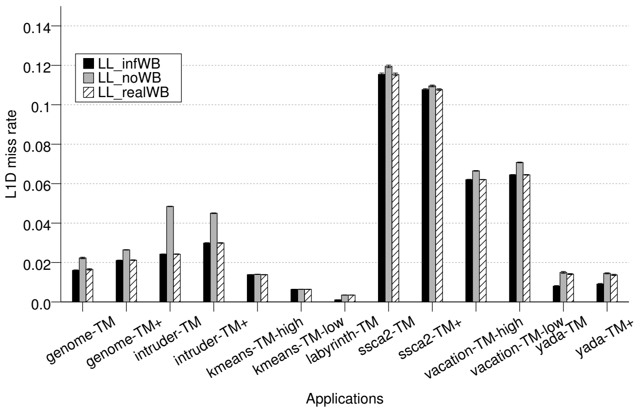


Fig. 3. Lazy design points: L1 data cache miss rates.

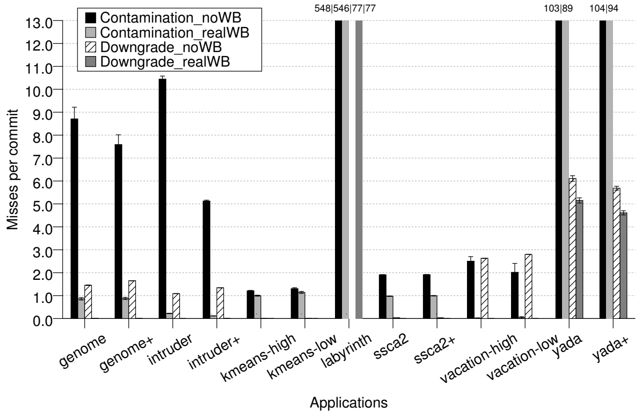


Fig. 4. Lazy design points: Downgrade and abort misses.

shows the average number of downgrade misses per committed transaction. We see reductions in both metrics when buffer sizes are large enough to contain most of the writes.

Fig. 5 shows the execution time breakdown of all applications, normalized to the execution time of the configuration with no write buffer, running 16 threads. The execution time is broken down into eight components – *barrier* is a measure of the time spent waiting at barriers; *non-txnal* corresponds to the number of cycles spent executing non-transactional code; *tx-useful and tx-aborted* represent cycles spent in transactional execution, split into useful and aborted cycles, respectively; *stall* is the time a transaction spent stalled in a data access, because such data was in the write set of a committing transaction; *backoff* represents the wait before an aborted transaction restarts, determined using a linear-backoff algorithm; *arbitration* and *acquisition* represent the overheads experienced at commit time, due to arbitration for the commit token and acquisition of exclusive ownership over modified lines, respectively. The figure shows a consistent improvement in performance when write buffers are present. Fig. 6 zooms in on the commit overheads, represented by the sum of cycles spent arbitrating for commit and acquisition of the write set, imposed by various configurations.

Genome. This workload runs high contention transactions in its first phase. Write buffers prove sufficient and are able to substantially mitigate abort misses, resulting in substantial improvements in L1 performance (seen in Fig. 3). In the latter phases, contention is relatively low and reductions in downgrade misses (see Fig. 4) provide further performance boost. Prefetching lines targeted by buffered speculative writes (*realWB_pf*) also yields substantial benefits (8-10%) by overlapping latency for data transfer with useful execution.

Intruder. The improvement in L1 cache performance is the most significant in intruder – an application with high contention, a large number of transactions and a medium-sized write set (about 50 bytes spread across 6 cache lines on average for its main transaction). Here, the impact of contamination in the private cache due to speculative writes is considerable. As described in Section II, repeated aborts cause invalidations of speculatively dirty data, which then result in misses when the transaction re-executes. Fig. 4 clearly shows the number of abort misses suffered by restarted transactions is significant in intruder, with an average of 10 such misses until a (perhaps repeatedly) restarted transaction eventually commits. This causes severe degradation in the L1 cache miss rate. The use of a write buffer completely eliminates abort misses for this application, and effectively reduces its cache miss rate by 40%, as shown in Fig. 3, for both configurations with speculative write-buffering enabled. The improvement in L1 cache performance shortens the duration of the transaction and thus reduces its probability of conflicting with other concurrent transactions. The nett effect is a substantial decrease of the number of aborted transactions (from almost 14000 in *noWB* to around 12200 in *realWB/idealWB*). This explains reductions in both *tx-aborted* and *backoff* components of the total execution time.

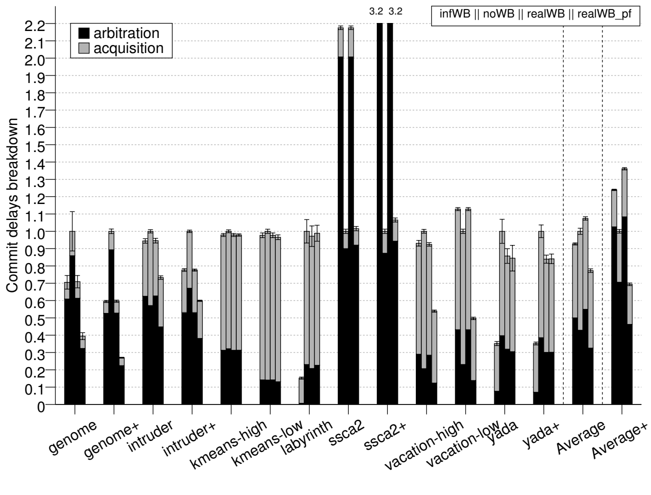


Fig. 6. Lazy design points: Commit and arbitration delays.

Kmeans. This application spends little time executing trans-

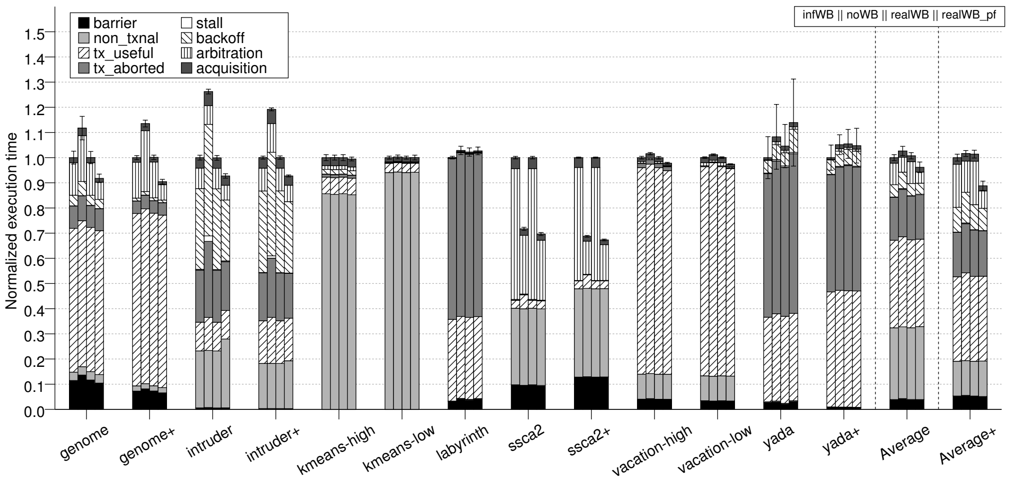


Fig. 5. Lazy design points: execution time breakdown.

actions. Moreover these transactions are tiny and behavioural variations between different configurations do not substantially impact performance. Nevertheless, write buffers achieves minor reductions in the number of abort misses (see Fig.4).

Labyrinth. In this benchmark, each thread replicates the global grid into its thread-local memory, and then applies Lee's routing algorithm on a local grid. Every time a thread creates a copy of the global grid, the cache lines that contain the local grid are likely to be still in modified state since the last commit, and thus must be written back to the L2 as well as downgraded to shared state before being speculatively modified again. At commit time, the writes to the local grid are indistinguishable from those to the global structure, and hence result in a large number of redundant coherence requests. Transactions here have extremely large write sets running to several hundred cache lines. Finite buffers prove insufficient and execution times remain largely independent of buffering configurations. The same trend is seen in L1 cache performance and the number of abort and downgrade misses. The idealized infinite write buffer configuration coupled with extremely high L1 hit rate significantly reduces commit and arbitration delays as seen in Fig. 6.

SSCA2. This workload has a very large number of predominantly non-conflicting tiny transactions that stress commit bandwidth. Thus, configurations that are able to reduce communication at commit time perform far better (30%) . Since contention is low, prefetching lines that would eventually be updated at commit improves performance significantly (realWB_pf in Fig. 5). Here, *noWB* configuration is interesting too since it effectively behaves like the prefetch configuration, reading in lines that would be eventually written.

Vacation. This workload shows little contention. Small improvements in performance can be noticed due to reductions in abort and downgrade misses. Prefetching also provides marginal improvements in performance.

Yada. Yada exhibits a high degree of cache contamination in the configuration with coherent buffering, with 92 abort misses per each commit of a restarted transaction. However, its very large write set (60 cache lines on average for its main transaction, with 2124 bytes written) makes it impossible for the 128-byte write buffer configuration to contain any substantial number of writes. Yet, it can be seen that *infWB* and *realWB* configurations perform better than others. This is because they are able to avoid, to a certain extent, interference when write-write conflicts exist.

C. Eager HTM Results

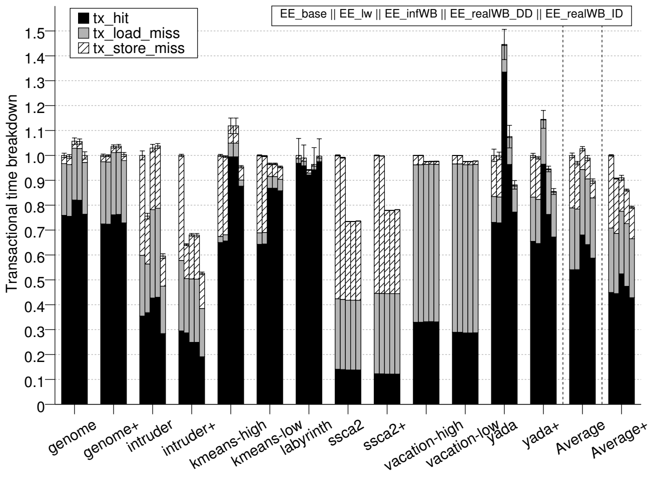


Fig. 8. Eager design points: transactional time breakdown

Fig. 7 compares execution times of the five eager configurations listed in II. The first four components are as described earlier; *stall_useful* represents stalls in cases when the transaction commits without aborting after the stall is released; *stall_aborted* represents stalls in cases when the stalled transaction eventually aborted; *rollback* represents cy-

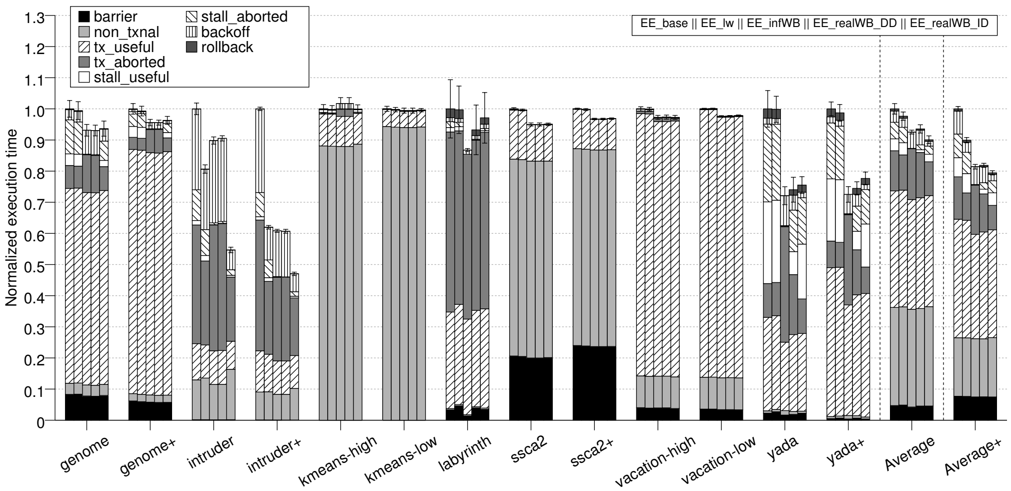


Fig. 7. Eager design points: execution time breakdown

cles spent in restoring old values upon abort. The figure shows write-buffering, and immediate draining in particular, improves performance significantly. Fig. 8 presents a breakdown transactional cycles spent waiting for memory accesses to complete. We discuss below important insights for each benchmark gathered from the data.

Genome. The early high contention phase in genome benefits from write-buffering. Transactions in this phase are relatively small so the size of the buffer does not matter much. Infinite buffering and deferred draining show marginal improvements over immediate draining (see Fig. 7). This is indicative of useful run-ahead execution past conflicts allowing readers to complete without stalling. An overall speedup of about 7% over the baseline is seen.

Intruder. This application benefits the most (more than 50% for intruder+) from introduction of write-buffering with immediate draining (Fig. 7). This is because conflicts are detected early resulting in stalls and consequently less demand for otherwise highly contended shared memory resources. Furthermore, immediate draining of writes behaves like prefetch of non-contended data. Deferred draining (and the infinite buffering case) causes injection of a lot of traffic into the network when transactions commit. This chokes directory resources and is unproductive since a large number of concurrently running transactions are conflicting. The overall impact of reduction in transactional execution time with immediate draining is magnified as it results in a contraction of the window of contention leading to greater concurrency. Cache performance is significantly better too (see Fig. 8). The lingering write optimization (EE_lw) also shows significant improvements (20% for intruder and 38% for intruder+) over the GEMS baseline implementation. Lingering writes are often re-activated when the restared transaction performs a new write to the same location, and in some cases effectively obtain lines in exclusive mode for transactional read-modify-write operations to complete successfully.

Kmeans. The application (both high and low contention variants) shows no major differences in execution time across different configurations. Transactions are tiny and most of the application is non-transactional. Minor (3%) degradation is seen with an infinite buffer or a finite deferred draining buffer when contention is high. This is because of the lighter commit time operations due to prefetch effects of immediate draining and reduced demands on directory banks at commit time.

Labyrinth. Finite buffering proves inadequate in this case. Deferred draining and infinite buffering manage to avoid unnecessary interference during execution time by mitigating write-write conflicts, achieving speedups of about 8-12%, partly because commits are typically fast due to the good cache performance over its write set (see Fig. 8).

SSCA2. Eager designs in general perform well for SSCA2 because of extremely low contention and high demands on commit bandwidth. Nevertheless, write-buffering manages to hide store latencies (Fig. 8 shows a 50% reduction in store miss times) and provides a 5% improvement that is consistent across all three buffering configurations. Prefetch effects of immediate draining are not visible because of tiny transactions.

Vacation. Its behavior is similar to that of SSCA2, with the exception that transactions are larger and fewer, and do not stress the system much. Write-buffering obtains marginal improvements (2-3%) in execution times, shown in Fig. 7.

Yada. Yada, like intruder, also shows significant improvements (30%) when write buffers are introduced. However unlike intruder, yada exhibits a significant number of write-write conflicts. These are hidden entirely in the infinite buffering configuration and to a smaller extent in the deferred draining configuration. Thus, these configurations perform 4-8% better than the immediate draining configuration.

D. Eager vs. Lazy: Relative performance

Fig. 9 compares the performance of three finite buffering configurations – EE_realWB_ID, LL_realWB_pf and

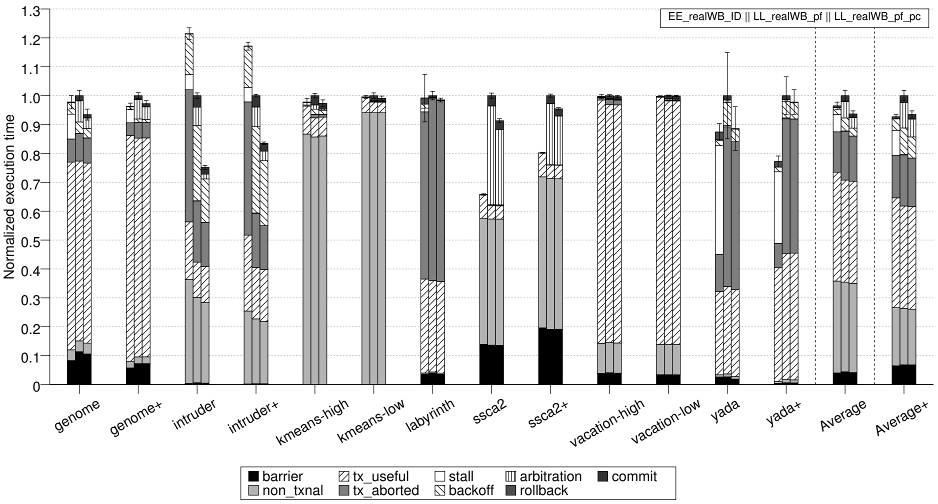


Fig. 9. Policy performance comparison: eager vs. lazy

LL_realWB_pf_pc. Overall we notice a leveling out of performance. These configurations are the best performing ones for each policy. Results in several previous studies consistently favor lazy designs. Here, we show that with the right buffering mechanisms, which are not related to TM policies at all, there is an equalization of performance. Lazy systems are still better in a highly contended application like intruder (by about 35%), but this is offset by significantly better performance by the eager design for SSCA2 (20-35% better). The eager design also shows substantial speedups over lazy for yada (12-22%). Overall, we see that the eager design performs as well as the lazy ones. This key insight highlights the importance of write-buffering in HTM systems, and shows that it is as important a consideration in HTM design as conflict resolution policy.

V. Related Work

This work presents a study of the impact of write buffering in HTM systems. The insights it encompasses apply to a large body of work done on HTM designs and yet are, to an extent, orthogonal to it. Concepts first described by Herlihy and Moss [9] have been leveraged by various HTM policy implementations.

Lazy Designs. The TCC proposal [8] implements buffering in a private cache with global commit arbitration using a bus. A committing transaction writes back all its speculative updates to the shared memory hierarchy. This broadcast is also used to keep the private caches coherent in a transactions-all-the-time framework. A later proposal [3] provides a commit algorithm which allows for considerable parallelism in directory based DSM systems. It works by dividing the directory into several banks. Transactions can commit in parallel if they do not observe directory bank conflicts. Commit sequence numbers are assigned to prioritize transactions when such conflicts occur. Tomic et al. [17] describe an eager conflict detection design that commits transactions lazily, utilizing directory coherence in MESI based systems with two levels of private caching. Negi et al. developed a broadcast-based lazy commit protocol in [13] that eliminates the need for write-backs or cache-line invalidation messaging at commit.

Eager Designs. LogTM [19] describes a protocol where transactional stores update memory in-place and store old values on the side. The design leverages coherence to implement conflict resolution and isolation. Bobba et al. [1] present a discussion of pathogical cases encountered in eager and lazy designs. Subsequent optimizations like signature based LogTM and FASTM [10] improve upon buffering and abort handling capabilities by extending coherence protocols.

Shriraman et al. [15] performed a comparative study of contention management policies in a hybrid FlexTM [16] based design. This study claims that systems lazy contention management achieves higher performance better than those with eager management. We would like to emphasize that conflict resolution policy is a factor that contributes towards overall HTM system performance but it is not the sole one. Effective management of updates in write buffers can tip the scales. Sanyal et al. [14] proposed schemes, involving both paging hardware and the operating system, to manage thread-local data separately to ease the burden on speculative versioning mechanisms. In this work, write buffers can achieve similar effects when they are large enough to capture most updates. Caches are not contaminated by speculative updates and commits and aborts do not penalize accesses to thread local data. Dahlgren et al. [4] analyzed the efficacy of write caches in parallel architectures supporting relaxed consistency models and demonstrated major improvements in miss penalties associated with coherence misses. While the study is not directly related to TM, the results therein suggest that transactional semantics permit flexibility in handling updates issued within atomic code blocks. Dice et al. [5] mention the use of store buffers to confine transactional updates in the Rock processor which provides limited support for TM constructs.

One common characteristic of HTM proposals mentioned in this section is that they do not investigate different write management mechanisms which can cause significant variation in key performance metrics. Write buffers are either absent or not clearly described. Considering the sensitivity of performance to the structural optimizations highlighted in this paper, ascribing improvements in performance metrics to changes in policy or high level protocol design is fraught with the risk of imprecision and oversimplification. This is of particular importance in scalable network-on-chip hierarchies where communication delays can be the major determinant of performance and we wish to emphasize the importance of accurate modeling when considering the complex interaction of multithreaded code with synchronization mechanisms in hardware.

VI. Conclusion and Future Work

In this work we have described and analyzed the inefficiencies that can be caused by buffering of speculative writes in coherent structures like private caches. While we do not recommend exclusive use of write buffers for managing transactional data as area and power restrictions may severely limit its utility, the importance of having such buffering to support the common case efficiently has been underlined. The performance impact of write-buffering has been quantified and shown to yield significant improvements in the set of benchmarks analyzed here. The expectation is that TM programming constructs would eventually enable workloads with coarse grained transactions, where non-contended data could be written along with actively contended data. Without appropriate write buffer support, in high contention scenarios abort misses would result in significant degradation of cache performance in both eager and lazy designs. In low contention but high commit throughput scenarios downgrade misses might result in substantial slowdowns in lazy designs due to prolonged arbitration. Moreover, when write buffers are present eager designs benefit from both the capability to hide store latencies and reduced logging actions. Shortened transactional execution times reduce the window of contention and improve overall performance. Another interesting insight is that write buffers bridge the performance gap between eager and lazy designs when contention is high, thereby indicating possible routes for the development of a general purpose, low complexity, high performance HTM architecture. We hope this will serve as a useful guide to architects planning to integrate hardware support for transactional memory in their designs.

Acknowledgment

Anurag's work at Chalmers has been supported by the European Commission FP7 project VELOX (ICT-216852). Rubén's work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14 475-C04. Rubén has a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152).

References

[1] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proc. of the 34th Int'l Symp. on Computer Architecture*, pages 81–91, Jun 2007.

[2] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symposium on Workload Characterization*, pages 35–46. Sept 2008.

[3] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *Proc. of the 13th Symp. on High-Performance Computer Architecture*, pages 97–108, 2007.

[4] Fredrik Dahlgren and Per Stenström. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 26:193–210, April 1995.

[5] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th Int'l Symposium on Architectural Support for Programming Language and Operating Systems*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[6] Aleksandar Dragojevic and Rachid Guerraoui. Predicting the scalability of an stm. In *TRANSACT '10: 5th ACM SIGPLAN Workshop on Transactional Computing*, Feb 2010.

[7] Sridhar Gopal, T. Vijaykumar, James S. Smith, and Guri Sohi. Speculative versioning cache. In *Proc. of the 4th Symp. on High-Performance Computer Architecture*, pages 195–206, 1998.

[8] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[9] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, pages 289–300. May 1993.

[10] Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proc. of the 18th Int'l Conf. on Parallel Architectures and Compilation Techniques*, Sep 2009.

[11] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[12] Milo M.K. Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa Alameldeen, Kevin Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, Sept 2005.

[13] Anurag Negi, M.M. Waliullah, and Per Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *Proc. of the Intl. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2010)*, pages 231–240, July 2010.

[14] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *HPCC '09: Proc. 11th Conference on High Performance Computing and Communications*, jun 2009.

[15] Arrvindh Shriraman and Sandhya Dwarkadas. Refereeing conflicts in hardware transactional memory. In *Proc. of the 23rd Int'l Conf. of Supercomputing*, pages 136–146, 2009.

[16] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proc. of the 35th Int'l Symp. on Computer Architecture*. Jun 2008.

[17] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proc. of the 42nd Int'l Symp. on Microarchitecture*, 2009.

[18] M.M. Waliullah and Per Stenstrom. Classification and elimination of conflicts in transactional memory systems. TR 2010:09, Dept. of Computer Science and Engineering, Chalmers University of Technology.

[19] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th Symp. on High-Performance Computer Architecture*, pages 261–272, Feb 2007.