π-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory

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

Abstract

Lazy hardware transactional memory has been shown to be more efficient at extracting available concurrency than its eager counterpart. However, it poses scalability challenges at commit time as existence of conflicts among concurrent transactions is not known prior to commit. Non-conflicting transactions may have to wait before committing, severely affecting performance in certain workloads. Early conflict detection can be employed to allow such transactions to commit simultaneously. In this paper we show that the potential of this technique has not yet been fully utilized, with design choices in prior work severely burdening common-case transactional execution to avoid some relatively uncommon correctness concerns. The paper quantifies the severity of the problem and develops π-TM, an early conflict detection - lazy conflict resolution design. This design highlights how, with modest extensions to existing directory-based coherence protocols, information regarding possible conflicts can be effectively used to achieve true parallelism at commit without burdening the common-case. We leverage the observation that contention is typically seen on only a small fraction of shared data accessed by coarse-grained transactions. Pessimistic invalidation of such lines when committing or aborting, therefore, enables fast common-case execution. Our results show that π-TM performs consistently well and, in particular, far better than previous work on early conflict detection in lazy HTM. We also identify a pathological scenario that lazy designs with early conflict detection suffer from and propose a simple hardware workaround to sidestep it.

1 Introduction

Hardware Transactional Memory (HTM) [7, 8] aims to bring the speed of silicon to programming constructs that offer freedom from locks by providing optimistic concurrency control for multithreaded applications. Shared memory accesses within sections of code, demarcated as atomic blocks or transactions, are tracked and managed by hardware mechanisms, providing *atomicity* (the entire execution appears to have occurred without any intervening changes in global state) and *isolation* (changes made by a transaction are not accessible to other threads until it has committed, i.e. completed its work by publishing changes to the global state) at the granularity of transactions. These mechanisms also ensure forward progress and impose a globally consistent view on the ordering of transaction commits.

Hardware for HTM typically comprises support for processor checkpoints, mechanisms to track speculative accesses and protocols to detect and resolve conflicts. In situations when a transaction's execution is deemed unsafe, it is aborted i.e. all changes made by the transaction are undone and the transaction is re-executed after restoring processor state to what it was when the transaction started execution. Data races, which are indicative of unsafe executions, can be resolved either eagerly, i.e. when they happen between two or more concurrently running transactions, or lazily, i.e. when a committing transaction publishes its updates en masse. Lazy resolution allows concurrency in more scenarios than eager; in particular, when a race exists between a reader-transaction and a writer-transaction, execution of both transactions runs past the conflict in a lazy HTM, thereby permitting the possibility of a safe interleaving when the reader-transaction commits before the writer-transaction. On the contrary, eager HTMs must adopt a conservative approach here, resolving the conflict, upon discovery, in favor of one of the transactions.

Lazy HTMs are, therefore, theoretically more efficient than eager designs [15]. Yet, the necessity of en-masse publication of updates at a transaction-commit event raises issues of scalability. Quite often several concurrent non-conflicting transactions attempt to commit simultaneously. Simple schemes, like the acquisition of a global commit token, take a conservative approach and preclude simultaneous commits of such transactions. This severely limits performance in workloads with light contention and a fairly large number of transactions that demand a high

978-1-4673-0826-7/12/$26.00 ©2011 IEEE

commit bandwidth. More elaborate schemes like Scalable TCC [4, 13] attempt to provide a degree of commit parallelism when transactions target different directory banks. Although they leverage the coherence protocol, information in transactional coherence messages that may indicate contention is not utilized effectively with only commit-time invalidations are relied upon to abort conflicting transactions. Designs like FlexTM [15], LV*[12] and EazyHTM [16] provide lazy conflict resolution by recording conflicts as they happen, using this information to enable distributed commits. FlexTM chooses to do so in software and sacrifices progress guarantees to gain greater parallelism. Performance costs associated with software intervention and software verification challenges without watertight forward progress guarantees could limit the value of this approach. LV* is restricted to broadcast-based systems. EazyHTM, on the other hand, provides parallel lazy commits in hardware and ensures forward progress, but, as we show next, trades off common-case performance to achieve it. This crucial design aspect of commit scalability forms the crux of the problem we tackle in this work which, in our opinion, is the next logical step in the evolution of lazy HTM design.

Enabling parallel lazy commits in hardware requires correct handling of all possible interleavings of transactions. Figure 1-(a) shows what might happen if a buggy early conflict detection protocol does not enforce atomicity correctly. A transaction T1 that has updated cache lines A and B initiates commit operations, aborting transaction T2 in the validation phase. T2 now restarts and reads the new value of line A since T1 has acquired it exclusively. However, it is possible for T2 to read the old value of line B and potentially commit this unsafe execution, resulting in an atomicity violation. A trivial solution to overcome this issue requires every transaction to send an indication to the directory whenever the first read or the first write happens to a cache line, even though the line may be present in the private cache. In the example above, as shown in Figure 1-(b), T2 will send a message to the directory which will then be forwarded to the committer, T1. T1 will then ask T2 to retry the access till it has acquired exclusive ownership over line B. Thus, every first read and first write to any cache line accessed by a transaction is effectively a miss in the private cache. This turns out to be wasteful, particularly when contention is low or when large coarse-grained transactions abound.

Some careful thought reveals that the trivial approach described above, followed by EazyHTM [16], does not go far enough when utilizing the information available from coherence messages regarding potential data races. Records of conflicts are maintained only at the granularity of cores. Doing so leaves only three ways of dealing with a transaction's read set when aborting - first, wait for lines to be invalidated; second, invalidate the entire read-set; third, as described above, requires protocol support for delaying the re-executing transaction so that it is able to read the correct version of a conflicting cache line. The first option demands communication between cores and extensive protocol support. The second and third options are too conservative and represent equivalently severe penalties in common-case scenarios, limiting scalability due to excessive communication with the directory.

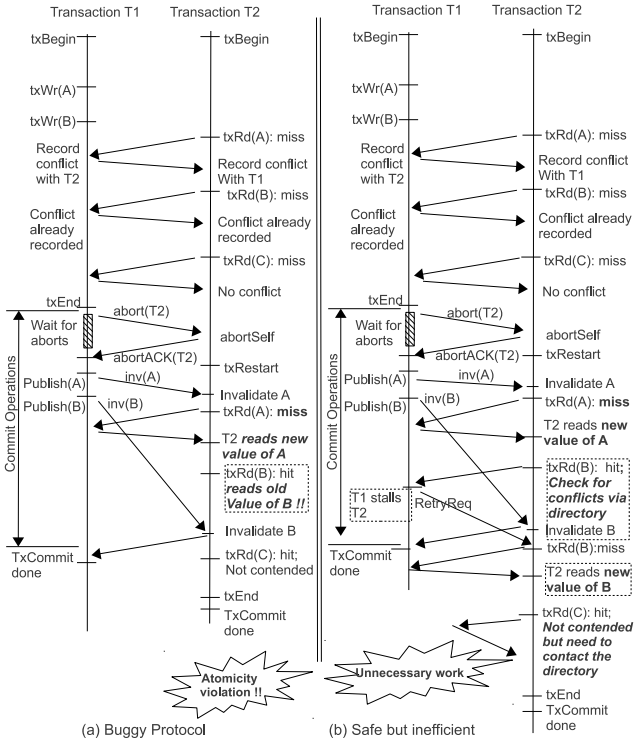


Figure 1. Problems with detecting conflicts early in lazy designs.

In this work we present a novel solution that achieves scalable lazy commits – π-TM. It consists of a few simple extensions to a directory-based coherence protocol and uses mild pessimism in the uncommon case to keep common case transactional execution unencumbered. As described later in Section 2, it works by recording conflicts at the granularity of cache-lines, simply by maintaining an additional single-bit annotation, called the π-bit, at the private cache. On an abort, only those cache-lines in a transaction's read-set that have seen a conflict during the course of its execution are invalidated along with speculatively updated lines. The transaction can now immediately restart and can use private cache data without any need to contact the directory. The committing transaction can now update shared system state in a much simpler fashion since the protocol ensures that no other private caches contain cache lines in its write-set. For safety, the committing transaction also invalidates any contended lines in its read set. We shall

show later the number of lines that actually face contention in large, coarse-grained transactions is *far smaller* than the total number of lines accessed in such transactions. Moreover, this method works equally well for small, fine-grained transactions. We believe that this solution overcomes a major performance bottleneck in scalable, lazy designs that make use of coherence protocols to resolve conflicts. The idea is reminiscent of the approach taken to reduce cache coherence overheads in Dynamic Self Invalidation [9].

The paper, in Section 4, presents an in-depth analysis of the impact of using π-TM. The study suggests that major benefits can be expected in low contention workloads and those with large transactions with moderate contention. High contention workloads with small transactions, like intruder from the STAMP benchmark suite [2], however, suffer due to the necessity to detect conflicts before a transaction can commit. In order to sidestep the pathology, this paper extends the π-TM design by incorporating local adaptability that permits transactions to switch independently between early and lazy conflict detection mechanisms. The results confirm that this simple artifice provides large improvements in high contention workloads while retaining the advantages of parallel commits in other scenarios.

2 Using coherence for conflict resolution

2.1 Prior work and unresolved problems

Cache coherence protocols allow HTM implementations to detect conflicts among concurrently running transactions at the granularity of cache lines. For most transactional workloads this represents a good trade-off between design cost and performance. Hence, it is no surprise that most scalable HTM design proposals choose to leverage coherence mechanisms for conflict resolution. In particular, eager conflict resolution protocols like LogTM [17] fit very naturally onto a cache-coherent CMP substrate, needing only modest design extensions to support TM semantics.

Lazy conflict resolution protocols, however, require protocol extensions to permit the existence of multiple speculative copies of a cache line while retaining the ability to use coherence messages to detect conflicts. This can typically be done in two ways. The *first method* involves containment of speculative writes within the private cache until commit. At commit time such writes are completed in shared memory causing conflicts to be detected at all concurrently running transactions that have accessed the line. The simplest way to do this is to allow only one transaction to commit at a time through the acquisition of a global commit token by the committing thread [6]. This is rather pessimistic resulting in marked performance degradation in scenarios when a large number of non-conflicting concurrent transactions exist and compete for commit permission. Scalable TCC [4] attempts to do better by letting commits targeting different directory banks proceed in parallel. This improves performance to an extent, as we shall see later, but does not provide true commit parallelism seen in designs like LogTM.

The reason for limited parallelism at commit time is that the committing transaction has no knowledge of which other concurrent transactions must abort to preserve atomicity. Thus, either commits must proceed one by one or complex commit protocols [4, 13, 14] must be employed to provide a degree of parallelism. In a cache coherent design this turns out to be inefficient. Information pertaining to potential conflicts is readily available from coherence messages suggesting a *second method* of performing lazy conflict resolution. This information, if retained over the course of execution of a transaction, permits true commit parallelism, since all potentially conflicting transactions that must be aborted would be known. All committers that have no races among themselves can then be sure that they can safely commit in parallel.

EazyHTM [16] describes a way to do so by maintaining two bitmaps – *killers list* (indicating transactions that can abort the local one) and *racers list* (indicating transactions that must be aborted on commit to preserve atomicity). It uses a special messaging protocol and hardware to indicate transactional accesses to cache lines to all potential sharers. Every first read to a cache line in a transaction requires all sharers be notified and any potential conflicts indicated before the transaction can safely proceed. A similar action is required for every first write to a cache line. It is quite clear that this action, although ensuring safety in the infrequent case described in Figure 1, represents a heavy burden on common-case transactional execution.

Can we do better? We note that coherence messages carry information that can not only be used to detect which core might be conflicting but also to figure out the cache line address of the contended line. This information can be recorded using a single bit cache line annotation in each private cache, which can later be used to resolve conflicts in a far more efficient manner. To investigate this idea further, in Section 2.2 we describe in detail the implementation of the baseline - early conflict detection employing concepts developed in [16] in a distributed directory based CMP. Section 2.3 then describes our proposal to extend this design with simple mechanisms to record contention at cache line granularity and resolve conflicts safely without burdening common-case transactional execution.

2.2 Baseline: Design and Operation

The architectural substrate for developing our approach consists of a mesh based network-on-chip (NOC) where each node in the mesh comprises a processing core, one level of cache private to each core, a slice of shared, inclu-

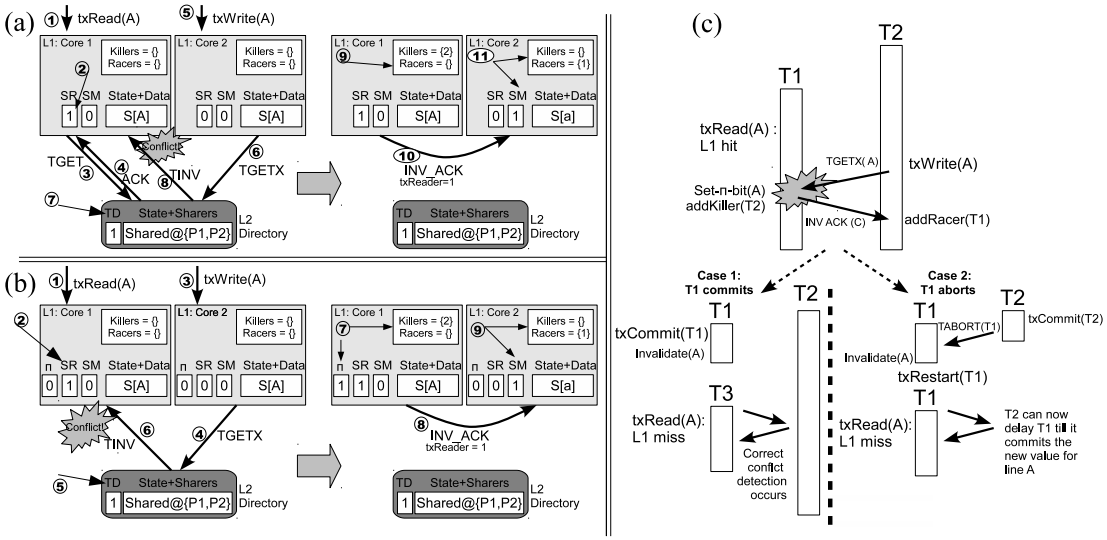


Figure 2. π-TM: Key protocol actions. (a) Baseline early conflict detection (b) Conflict detection in π-TM (c) π-TM: high level behavior

sive L2 cache with corresponding directory entries employing a MESI protocol and routing logic.

We retain the basic flow of conflict information between concurrent transactions as described in [16]. When a line is read by a transaction any concurrent transactions that might have written the line must be notified. When a line is written by a transaction all sharers must be notified that a new potential conflict might exist. The requester must also be made aware of which concurrently running transactions have conflicts with it. We choose to piggy-back this information on usual coherence messages (TGET, TINV and various ACKs) using two single-bit flags – *txReader* for transactional read, *txWriter* for transactional write. This is simpler than having a host of new protocol messages as used in [16]. Moreover, in our opinion most TM use-cases do not justify investment of hardware resources into ad-hoc communication mechanisms solely dedicated to conflict detection. Thus, the baseline leverages the NOC communication fabric for all information exchange, both transactional or non-transactional. Conflict detection involves 3-way communication – *(1) TGET (with txWriter/txReader) → directory (2) TINV → sharers (3) ACKs from sharers → requestor.*

The directory maintains a TD bit (transactionally dirty) as proposed in [16], which when set indicates the presence of one or more speculatively modified versions of the line. This allows the directory to eliminate unnecessary communication with sharers when no conflicts exist. The directory is capable of resolving the source for forwarded data in case of transactional reads. This could be the L2 (if the line is not the Modified state at some core) or a private L1 cache (containing the consistent up-to-date copy). Handling of accesses at the directory is summarized in Figure 3. Reception of *non-transactional* invalidations causes aborts if a commit is not already in progress. Transactional invalidations targeting lines that have not been transactionally accessed locally cause such lines to be invalidated.

Figure 2-(a) depicts how the baseline protocol operates. Transaction *T1* on core 1 wishes to read cache line *A* (step-1) which is in "S" state in its private cache. Core 1 sets the SR flag for the line and sends a *TGET(A)*, with *txReader* high, request to the directory (steps 2,3). The directory finds the *TD-bit* not set and responds with an *ACK* allowing *T1* to proceed (step-4). Subsequently, transaction *T2* on core 2 attempts to write line A and sends a *TGET(A)* request, with *txWriter* set, to the directory (steps 5,6). The directory sets the TD-bit for the line and sends a TINV(A), with *txWriter* set, to core 1 (steps 7,8). Core 1 notices the conflict, adds core 2 to the killer list and sends an *INV_ACK* acknowledgment to core 2 with the *txReader* flag set (steps 9,10). Core 2, on receiving the acknowledgment knows that *T1* is a racer and adds core 1 to its racers list, completing the conflict detection protocol. Before it publishes any of its updates on commit, *T2* will abort *T1* by sending a *TABORT* message to core 1. The rare case of racing commits is handled by using a unique core identifier as priority.

2.3 π-TM: Design and Operation

We build on the baseline described earlier by adding a new *π-bit* to existing private cache lines. When a transac-

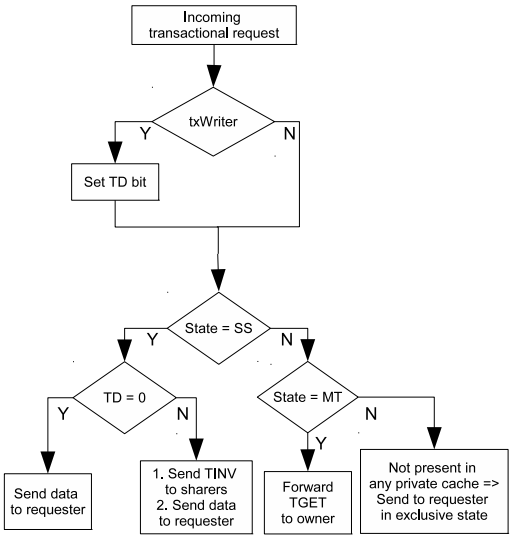


Figure 3. Access handling at directory

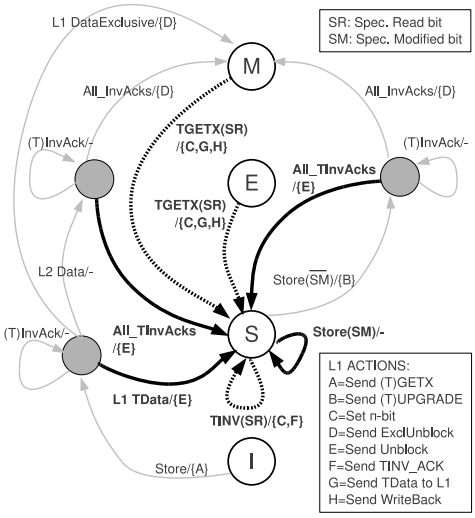


Figure 4. Supporting transitions at L1

tion that has only read a line finds the line contended, it sets the corresponding π-bit in its private cache. Lines with the π-bit set are invalidated when the transaction commits or aborts. We term this as *pessimistic invalidation* (hence the name π-*TM*) since all *possibly conflicting* lines are invalidated rather than invalidating only those lines that are being written by the committing transaction. The rationale behind this approach is that the number of such lines (speculatively read and contended) is typically small (see evaluation) and the cost of this minor pessimism is far outweighed by performance advantages provided by unencumbered common-case transactional execution. Invalidation of such lines at commit is required to preserve atomicity. This prevents a subsequent transaction from reading an old value from the contended line and newly committed value from a different line, when both lines belong to the write-set of another committing transaction. Note that abort messages from cores not indicated as killers in the *killers-list* are ignored.

Invalidation of lines with π-bit set guarantees that all valid lines in the private cache can be safely read by transactional code without contacting the directory. Since lines that have not been accessed transactionally are invalidated by incoming *TINV* messages, no unsafe accesses can be handled locally. Lines in the read set could also be invalidated at the point when contention is noticed, as it is simple to track the read set with a signature [3]. This invalidation is inefficient as such lines could be accessed again in the transaction. More importantly, it just tackles one part of the problem by ensuring safety only in the case where a line is speculatively read before contention is noticed.

Figure 2-(b) depicts key protocol actions when pessimistic invalidation is used. Core 1 running transaction *T1* reads a line, *A*, in its cache, setting the corresponding *SR-bit* (steps 1,2). Note that no communication with the directory occurs. The line is subsequently written by core 2 running transaction *T2* which attempts to acquire exclusive permission over the line by sending a *TGET(A)*, with txWriter set, to the directory (steps 3,4). The directory now sets the *TD-bit* and sends a *TINV(A)*, with *txWriter* set, to core 1 (steps 5,6). Core 1 recognizes the conflict and adds core 2 as a killer setting the π-bit for the line (step 7). It then sends an ACK to core 2 with *txReader* flag set, allowing core 2 to add core 1 to its racers list (steps 8,9). This completes the π-TM conflict detection protocol in this example.

Now, two scenarios (shown in Figure 2-(c)) can arise based on whether or not *T1* commits before *T2* aborts it. If *T1* commits before *T2* can abort it, the line with the π-bit set is invalidated and the thread can continue further execution. If *T2* commits before *T1*, core 2 sends a *TABORT* message to core 1. Since core 2 marked as a killer, *T1* aborts and invalidates contended line. In any scenario, the possibility that core 1 later accesses line *A*, a contended line, without communicating with the directory (thereby preventing conflict detection) is eliminated. A key point to be noted here is that a speculatively written line (with SM-bit set) may be read by the transaction without any communication to potential killers. Thus, lines in the write-set are always reported as *both read and written* when conflict detection acknowledgments are sent.

Figure 4 summarizes new protocol transitions required at the private cache controller. Transitions in bold black lines support lazy versioning. Those depicted using dashed black lines allow pessimistic invalidation of cache lines with π-bit set. The grey transitions are part of the standard MESI

coherence protocol (showing all pertinent transitions). Grey circles represent transient states for in-flight operations. No new coherence states are needed.

2.4 The Need for Adaptability

It is difficult to come across a one-size-fits-all HTM design. This is true both for eager resolution designs that perform poorly when contention is high and for lazy resolution designs that suffer when contention is low. Early conflict detection brings together the best of both worlds, but in doing so acquires a weakness not found in other designs. This situation arises when small to moderate sized transactions see high contention on late accesses that occur shortly before commit. In such cases the requirement to detect all conflicts before the transaction can commit brings the latency for doing so in the critical path. In our analysis later, we encounter this scenario in the application *intruder*, which might represent an entire class of future TM workloads. We also note that lazy conflict detection works well in this case since it can combine conflict detection with write-set publication. This suggests a way to build in adaptability into the system by detecting the condition and switching policy to achieve higher performance. We choose the simplest lazy conflict detection protocol – the global commit token (*GCT*) approach [1] – to evaluate the idea.

Each atomic block in a thread can operate in one of two modes – π*-mode* or *GCT-mode* – independently of the rest. Since both modes confine speculative updates within private caches the two protocols can interact safely. A running GCT-transaction aborts when it receives a *TABORT* message from a π-transaction. A committing GCT-transaction aborts a validating π-transaction by responding to the *TABORT* message with *NACK*. Moreover, a GCT-transaction does not release its write-set until the entire commit is done, thereby preventing unsafe interleavings from occurring. When threads operate in mixed-mode, all π-transactions have accurate information regarding racers. However, a π-transaction may not know all its killers, since killer GCT-transactions do not indicate conflicts over their write-set. This is not a safety issue since such GCT-transactions will cause any racing π-transactions to be aborted by non-transactional invalidations.

Figure 5 shows structures needed to support this adaptability. These requirements are not very large. Transaction identifiers (e.g. instruction pointers or sequence numbers assigned to "*start-transaction*" instructions) are mapped to one of several predictors (we use 16 predictors). Hardware overheads are minor, as can be inferred from the figure. It should be noted that the decision to switch modes is taken entirely based on information locally available at each core.

The algorithm for switching (see Algorithm 1) relies upon two pairs of thresholds to switch between the two

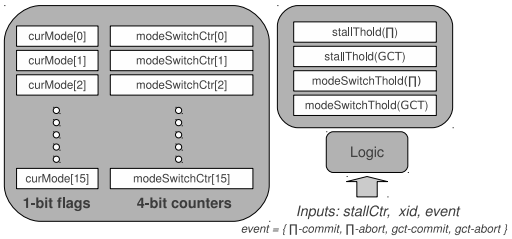


Figure 5. Hardware structures that enable mode switches.

Algorithm 1 Mode Switch Prediction Logic

```
if π-abort OR GCT-commit then
    if stallCtr > stallThold(curMode[xid]) then
        modeSwitchCtr[xid] ← modeSwitchCtr[xid] + 1
        if modeSwitchCtr[xid] > modeSwitchThold(curMode[xid]) then
            flipCurMode(xid)
            modeSwitchCtr[xid] ← 0
        end if
    else
        modeSwitchCtr[xid] ← max(0, modeSwitchCtr[xid] − 1)
    end if
else
    if π-commit OR GCT-abort then
        modeSwitchCtr[xid] ← 0
    end if
end if
```

modes of opertation. In π-mode, *stallCtr* represents the time between arrival at the end of a transaction and an abort before the initiation of validation operations. Consistently high stall counts and subsequent aborts indicate high contention and late conflicts – conditions which degrade early conflict detection performance. In GCT-mode, *stallCtr* represents the time spent in arbitrating for a global commit token before a successful commit. Consistently high arbitration stalls preceding successful commits indicate low contention and demand for commit bandwidth, i.e. conditions where π-mode works best. Since latencies depend strongly on network topology and available on-chip bandwidth, the two pairs of thresholds can be determined reasonably well at design time and need not be learned online.

3 Experimental Setup

Architectural Substrate. We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set [11], in conjunction with Virtutech Simics [10]. 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 ex-

periments on a 16-core tiled CMP system, as described in Table 1. The L1 caches maintain inclusion with the L2. L1 caches are kept coherent through an on-chip distributed directory (associated with L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol.

Table 1. System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Adaptable π-TM Thresholds	
$\pi \rightarrow GCT$ $GCT \rightarrow \pi$	2 consecutive 512-cycle stalls 4 consecutive 256-cycle stalls
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split 4-way, 1-cycle latency
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

Design Points. The performance of several HTM design points is now evaluated and compared on the parallel architecture described above. In addition to the early conflict detection design points already discussed – baseline, π-TM, adaptable π-TM – we developed detailed models for the following lazy conflict resolution design points - a global commit token (GCT) design and a detailed implementation of the Scalable TCC (STCC) design [4]. All evaluated configurations are summarized in Table 2. The GCT design is as described by Bobba et al. [1]. It buffers all speculative updates locally, acquiring exclusive permissions to commit before publishing updates to shared memory. This design models pessimistic lazy commits which sacrifice some performance for simplicity. STCC models in detail the commit mechanism described in [4]. It allows commit parallelism among transactions that target different banks of a distributed directory. We also model an eager conflict resolution HTM that closely follows the LogTM approach as described in [17]. All configurations employ a linear backoff algorithm before restarting an aborted transaction.

Table 2. HTM Configurations

Configuration	Description
Base	Baseline Early Conflict Detection [16]
PI-TM	Basic π-TM scheme
aPI-TM	Adaptable; can choose between π-mode or GCT mode
LL-GCT	Lazy conflict resolution using a global commit token [1]
LL-STCC	Scalable TCC [4]
EE	Eager conflict resolution based on LogTM [17]

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 [5]. 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. Results presented have been averaged over ten runs for each application, each with very minor randomization of some system parameters just sufficient to excite different interleavings. Command lines and parameters for these workloads can be found in [2].

Table 3. Execution time components

Component	Description
barrier	wait at barriers
non_txnal	non-transactional execution
tx_useful	successful transactional execution
tx_aborted	aborted transactional execution
stall	EE: conflicting request retries Lazy: delay between transaction end and sending of *TABORTs*
backoff	idle cycles after an abort
arbitration	acquisition of lazy commit permission
commit	cycles spent committing updates to shared memory
rollback	cycles spent aborting transactions

4 Evaluation

4.1 Early Conflict Detection Performance

Figure 6 shows the relative performance of the three early conflict detection designs – the baseline design, π-TM and adaptable π-TM. Each bar is composed of distinctly shaded regions indicating the distribution of execution cycles over various activities described in Table 3. Figure 7 shows the distribution of transaction handling modes (π or gct) in different workloads for adaptable configuration (aPI-TM). Figure 8 compares the miss-rates at the private caches over the duration of execution of each application.

Table 4 presents contention statistics for various STAMP workloads. The number of lines that are invalidated pessimistically constitutes a small fraction of the read-set size for most applications, corroborating choices made in the π-TM design. When transactions commit almost the entire read-set is non-contended. When transactions abort the number of invalidations typically amounts to less than 10% of the read-set size at commit. Only in the case of yada about 48% of the read-set is invalidated on aborts.

Therefore, in Figure 6 we see that π-TM is substantially

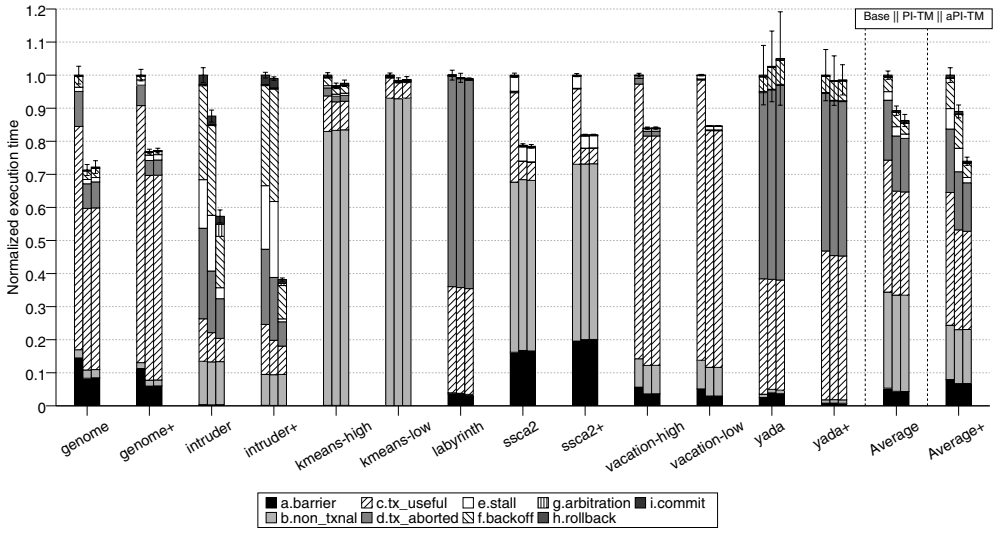


Figure 6. A comparison of early conflict detection schemes.

Table 4. Contention statistics.

	π-lines @abort	π-lines @commit	Rset (#lines) @commit	π-lines (% Rset) @abort	Killers @commit	Killers @abort	Racers @commit	Racers @abort
genome+	1.91	0.01	22.8	8%	0.01	1.91	0.07	1.17
genome	2.70	0.04	25.7	11%	0.03	2.87	0.18	2.23
intruder+	0.32	0.03	11.2	3%	0.05	3.11	1.53	4.01
intruder	0.54	0.07	9.1	6%	0.09	3.18	1.39	3.98
kmeans-high	0.63	0	5.4	12%	0	1.12	0.22	0.98
kmeans-low	0.65	0	5.4	12%	0	0.94	0.04	0.79
labyrinth	1.31	0.08	71.1	2%	0.11	1.33	0.63	1.02
ssca2+	0.07	0	3	2%	0	1.66	0	1.39
ssca2	0.11	0	3	4%	0	1.81	0.01	1.61
vacation-high	0.65	0.01	66.8	1%	0.01	0.97	0.03	0.03
vacation-low	0.60	0	53.9	1%	0	0.99	0.01	0.03
yada+	16.27	0.23	33.8	48%	0.12	4.46	0.37	4.4
yada	13.31	0.48	27.5	48%	0.25	3.89	0.63	3.7

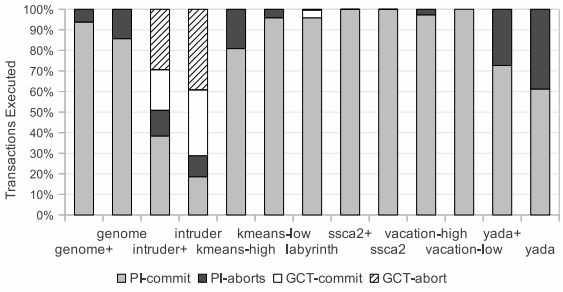


Figure 7. Adaptable π-TM: Mode distribution for transactions

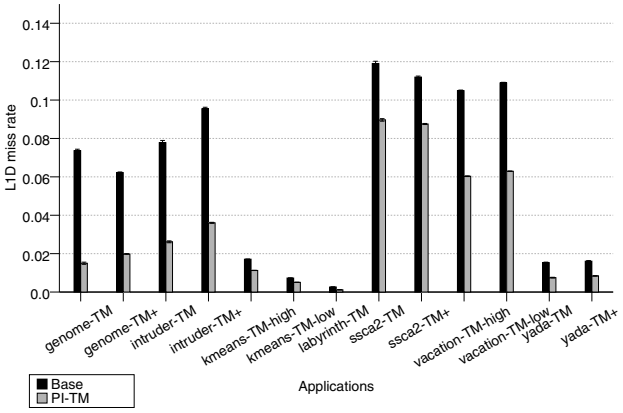


Figure 8. Miss rates

(more than 10%) better than the baseline. The adaptable variant (*aPI-TM*) performs the best (about 16% better than Base and about 25% better for long running workloads). In Figure 9 we see that π-TM achieves major reductions in network traffic, both in terms of flit count (about 15% less) and number of messages released into the network by the workload (about 20% less). In the CMP fabric used for simulations large on-chip communication bandwidth is available to each in-order processor. In systems with limited bandwidth, it can be expected that the lower demands of π-TM would translate into improved performance.

We now analyze individual workloads in detail highlighting aspects that influence performance.

Genome, SSCA2 and Vacation. Communication overheads induced by baseline early conflict detection mechanisms result in considerable degradation of private cache performance as can be seen in Figure 8. This translates into slower execution of transactions, a fact which is clearly highlighted by the significantly (almost 40% for genome) higher useful transactional execution time (*tx_useful*) component in the execution time breakdown shown in Figure 6. The performance between the basic π-TM design and its adaptable variant is little. This is because contention is relatively low for most of the application.

Intruder. Intruder exhibits high levels of contention. Small transactions with late updates to shared data compete. Though the basic π-TM design shows some improvement over the baseline on account of better cache performance, it suffers from the pathology described in Section 2.4. The requirement to complete conflict detection before commit operations can be initiated results in substantial increases in aborted execution and backoff time. The adaptable variant is able to switch policies quickly for offending transactions and avoids performance degradation, showing a performance improvement of 40-60 %. Figure 7 shows how the switch to GCT-mode occurs with about 50-70% of the transactions committing in the GCT-mode.

Kmeans. This application spends only a small fraction of its execution time (<10%) executing transactions. Moreover, these transactions are tiny. Therefore, we see no major deviations in performance although the two π-commit variants perform approx. 2% better than the baseline.

Labyrinth. This application has very large transactions where write sets run into hundreds of cache lines. Most such writes are also first accesses to lines, targeting the local grid, and hence, cache performance is very good, even in the baseline scenario. No major performance deviations are seen across all four configurations.

Yada. This mesh refinement algorithm exhibits moderate to high contention spread over a relatively large dataset. Although cache performance improves (by 50%, see Figure 8), this does not directly translate into improved performance when using the small input set. Execution times vary by 30% across different runs. When using the larger dataset, we see that the π-commit variants perform marginally better than the baseline (by approx 2%).

4.2 Comparison with other designs

Figure 10 shows the relative performance of the adaptable π-TM design and other pertinent design points. These include LogTM (shown as EE), a lazy global commit token approach (LL-GCT) and the scalable TCC design (LL-STCC). The execution times have been normalized to those achieved by the baseline configuration.

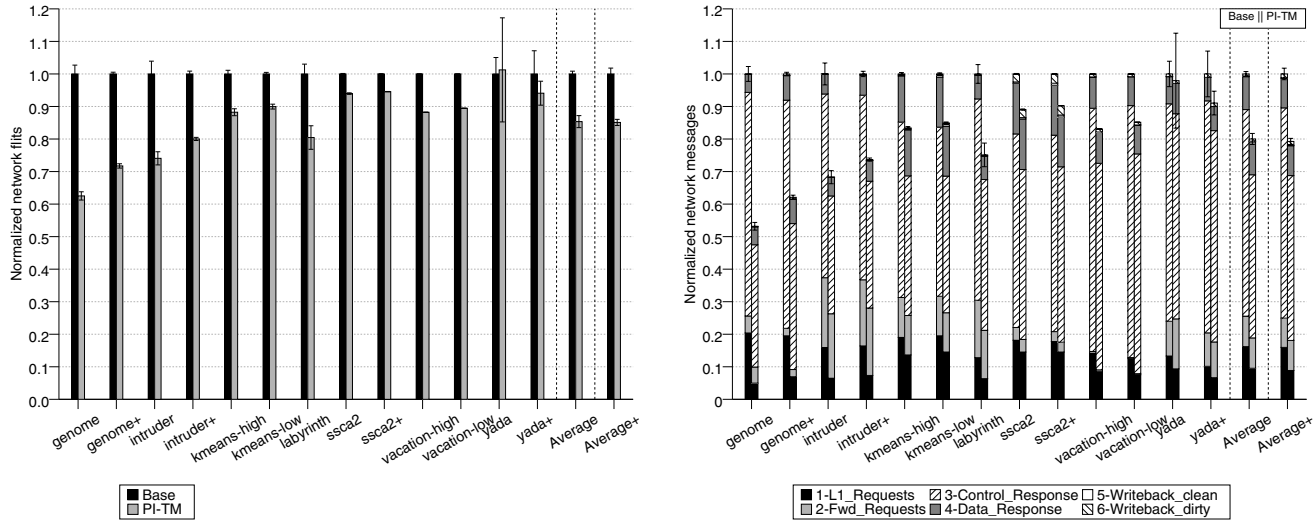
All four designs perform better than the early conflict detection baseline design. This is primarily because of the poor private cache performance of the baseline. Furthermore, we note that adaptable π-TM achieves the best performance overall. This design performs consistently well for all workloads, indicating robustness and the ability to avoid pathological scenarios. The difference between the lazy conflict detection designs and the adaptable π-TM can be seen in the reduced arbitration component. This clearly shows how parallel commits can be effective in improving overall system performance.

We now present some important workload-specific observations gathered from the data.

Genome. π-TM performs significantly better than all other designs. As we have explained earlier, this is primarily due to benefits of pessimistic invalidation under moderate contention and large transactions. As a result of contention for commit permissions, LL-GCT shows a large *arbitration* component in Figure 10 which is absent in the π-TM case. LL-STCC allows a degree of commit parallelism, but commit-time communication to the directory banks results in longer commit durations. This clearly highlights the efficacy of parallel commits in π-TM. The EE design detects conflicts on memory stores as they occur before proceeding with further execution. π-TM, on the other hand, is able to run-ahead past potentially conflicting stores overlapping the line-fetch with useful work.

Intruder. In this high contention application lazy designs come out on top. The ability to commit in parallel does not bring much advantage here as most transactions conflict. Early conflict detection mechanisms suffer from the pathology mentioned in Section 2.4. However, the adaptable π-TM variant sidesteps it by quickly switching problematic transactions to GCT-mode. The design is about 5% faster than LL-STCC in long running simulations (intruder+).

SSCA2. This application has very low contention but the very large number of tiny transactions stresses commit bandwidth in each design. Both variants of π-TM perform the best here, marginally (3-5%) better than the EE design. The reason, as seen in the case of genome, is the lack of latency hiding capabilities for stores in the EE design. Lim-



(a) Flit count

(b) Message Type Distribution

Figure 9. Network Traffic.

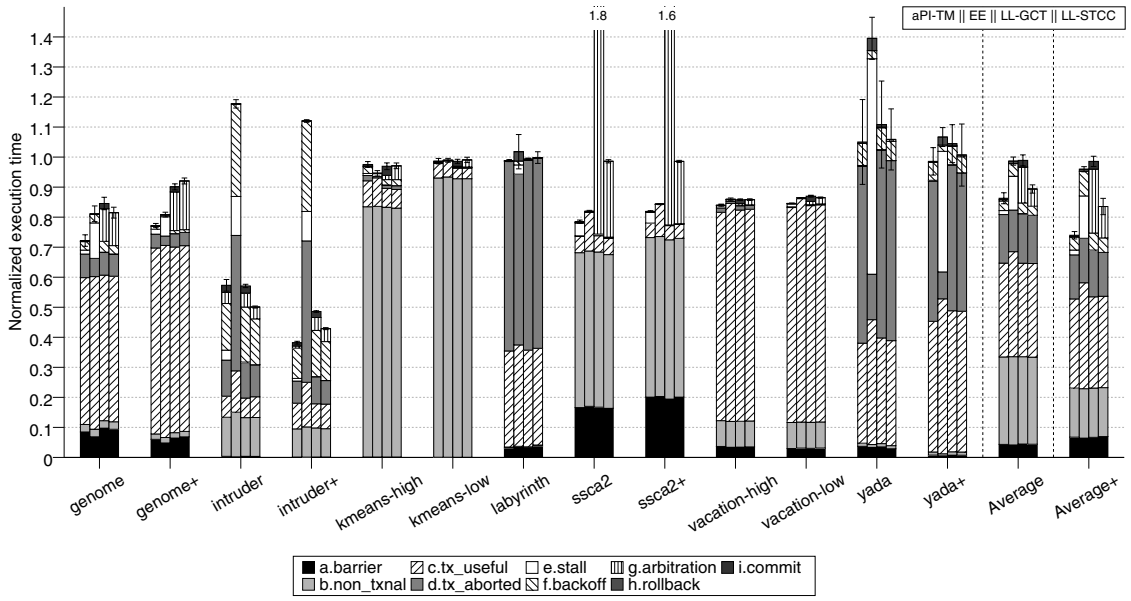


Figure 10. π-TM: Comparison with other designs.

ited commit bandwidth when using a global commit token results in pathological behavior. Scalable TCC mitigates this effect but is around 25% slower than LogTM or π-TM due to the commit-time communication with the directory.

Yada. Yada exhibits moderate contention spread over a relatively large dataset. The adaptable π-TM design performs as well as or marginally better than LL-STCC, taking full advantage of both commit parallelism (Figure 7 shows no GCT-mode transactions) and lazy execution. The EE design does not perform well and turns out to be the slowest (40% slower for yada and about 10% slower for yada+ when compared to the adaptable π-TM design). As seen before, the EE design suffers due to exposed latency for stores.

Kmeans, Labyrinth, Vacation. These applications show no major deviation in performance as transactions either contribute little to the overall execution cycles (e.g. kmeans) or the contention is low (e.g. vacation) or the cache performance is good (labyrinth).

5 Concluding Remarks

The design space of HTM systems has been studied extensively in recent research. Yet the quest for true optimism has been thwarted by inherent *pessimisms* in different TM policies. Traditional lazy designs are pessimistic when they attempt to commit. Eager designs are pessimistic when they detect conflicts. When implementing TM on a coherent CMP substrate information regarding conflicts is readily available and, if used well, leads to a design that can achieve the best of both worlds – parallel commits with optimistic lazy run-ahead past conflicts. Unfortunately, early work in this direction has reverted to pessimism in another form and in a rather more critical scenario – treating every fresh transactional access to a cache line as potentially contended. Our work shows this can degrade performance very substantially. The π-TM design drops this pessimism when it turns out to be prudent to do so, resorting to a far milder form of pessimism that leaves the common-case unburdened. The paper presents strong evidence that supports this claim. We also demonstrate the importance of incorporating adaptability in design to changing workload characteristics in achieving higher performance and improved robustness.

Acknowledgments

This collaborative 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. It was also partly supported by the HiPEAC-2 NoE under contract FP7/IST-217068. Anurag's work at Chalmers was supported by the European Commission FP7 project VELOX (ICT-216852). Rubén Titos, now a post-doctoral researcher at Chalmers, was supported by a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152).

References

[1] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. 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] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, pages 35–46. Sept 2008.

[3] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Int'l Symp. on Computer Architecture*, pages 227–238, Jun 2006.

[4] 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 *Proc. of the 13th Symp. on High-Performance Computer Architecture*, pages 97–108, 2007.

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

[6] 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 *Proc. of the 31st Int'l Symp. on Computer Architecture*, pages 102–113, Jun 2004.

[7] M. Herlihy and J. E. 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.

[8] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[9] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *In Proceedings of the 22nd Int'l Symp. on Computer Architecture*, pages 48–59, 1995.

[10] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[11] M. M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, Sept 2005.

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

[13] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 144–154, Oct 2008.

[14] X. Qian, W. Ahn, and J. Torrellas. ScalableBulk: Scalable cache coherence for atomic blocks in a lazy environment. In *In Proc. of the 43rd Int'l Symp. on Microarchitecture, MICRO 2010*, pages 447–458, 2010.

[15] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *Proc. of the 35th Int'l Symp. on Computer Architecture*. Jun 2008.

[16] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proc. of the 42nd Int'l Symp. on Microarchitecture*, 2009.

[17] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. 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.