Speculation-Based Conflict Resolution in Hardware Transactional Memory

Rubén Titos, Manuel E. Acacio, José M. García
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia
Murcia, Spain
{*rtitos,meacacio,jmgarcia*}*@ditec.um.es*

Abstract

Conflict management is a key design dimension of hardware transactional memory (HTM) systems, and the implementation of efficient mechanisms for detection and resolution becomes critical when conflicts are not a rare event. Current designs address this problem from two opposite perspectives, namely, lazy and eager schemes. While the former approach is based on an purely optimistic view that is not well-suited when conflicts become frequent, the latter results too pessimistic because resolves conflicts too conservatively, often limiting concurrency unnecessarily. In this paper, we present a hybrid, pseudo-optimistic scheme of conflict resolution for HTM systems that recaptures the concept of speculation to allow transactions to continue their execution past conflicting accesses. Simulation results show that our proposal is capable of combining the advantages of both classical approaches. For the STAMP transactional benchmarks, our hybrid scheme outperforms both eager and lazy systems with average reductions in execution time of 8 and 17%, respectively, and it decreases network traffic by another 17% compared to the eager policy.

1. Introduction

Transactional Memory (TM) is a technique that tackles the complexity of parallel programming by unburdening the developer from the intricacies of lock-based synchronization, yet produce efficient multithreaded programs. Using the TM model, the programmer wraps the critical regions of the code using *xact_begin*/*xact_commit* statements or *atomic* blocks, turning groups of accesses to shared data into transactions that the underlying levels must then execute atomically and without interference from concurrent threads.

In order to extract as much parallelism as possible, TM systems usually execute transactions optimistically, allowing multiple threads to run critical sections concurrently, while monitoring their execution in order to detect and resolve real run-time data conflicts whenever they occur. In order to identify such violations of isolation, a TM system must keep track of the addresses read and written by each running transaction. A conflict appears when two or more concurrent transactions access the same location and at least one of the accesses is a write, and the conflict management scheme is a key design point of a transactional system.

Although TM has been depicted as a promising way to smooth the transition to multicore systems, its success is yet to be demonstrated, making chip manufacturers very prudent about dedicating hardware to accommodate transactional mechanisms. The adoption of TM by the industry may be aided by evolutionary designs that incorporate these functionalities as a complement to the baseline chip multiprocessor architecture, enabling both transactional and non-transactional execution of parallel programs. Nevertheless, a necessary condition for the triumph of TM-capable chips is their ability to cope with a great variety of workload scenarios and deliver reasonable performance at all times. Transactional applications developed under constraints of programming productivity will expose very different features to those well-balanced, minimally synchronized parallel codes written by *heroic* programmers. Moreover, as developers assume that TM relieves them from part of their synchronization duties, it is expectable that some will indeed put less effort in writing conflict-free code. Considering the fundamental principle of the TM programming model – ease programming through coarse-grain synchronization –, TM systems should achieve decent performance levels regardless of the quality of the executed parallel code, at least to a certain extent.

Hence, if TM is to unburden the common programmer from the correctness/performance tradeoff imposed by locks, we must stop the assumption of *transactions are small and short running* from becoming a self fulfilling prophesy, as mentioned by Bobba et al. [2]. Nonetheless, encouraging the programmer to use coarse-grain critical sections is a double-edged sword that may lead to conservative synchronization and create artificially large transactions that increase conflict probability and cause unnecessary conflicts. To this respect, benchmarks composed of coarse-grain transactions have already shown that conflicts can be very frequent [18]. Therefore, within its possibilities, the hardware must do its best to extract parallelism from any given set of concurrent

transactions – no matter the number of conflicts that may arise – and it should do so in a completely transparent manner to higher abstraction levels. Just like superscalar processors include aggressive hardware to extract ILP *under the hood*, we believe transactional multicore chips should attempt to exploit all the *thread level parallelism* they can find, and do so hiding its intricacies to upper levels, so that hardware TM implementations can evolve over time without affecting the abstractions of the basic transactional components and their interfaces.

Hardware transactional memory (HTM) systems proposed to date use two distinct strategies for conflict management. Systems that detect and resolve conflicts as soon as possible are called eager, while those that delay conflict management until commit time are referred as lazy. When executing large transactions that modify highly-accessed shared variables, the optimistic approach of lazy HTMs may result counter-productive because these transactions are likely to conflict with other concurrent readers, discarding proportionally large amounts of valid computations, as shown by Figure 1 (b). On the other hand, an eager policy finds conflicts early and tries to solve them through stalls rather than drastic aborts, in an attempt to reduce the amount of wasted work by conflicting transactions – Figure 1 (a) –. In spite of that, the eager strategy is often too pessimistic because it keeps isolation of transactional data until commit, serializing the execution of the conflicting transactions.

In this paper, we propose a hybrid conflict management scheme that combines the benefits of eager and lazy policies in order better accommodate large, conflict-prone transactions. This novel approach inherits most of its features from eager HTM designs, while it emulates the behaviour of lazy systems by employing speculation and commit order enforcement to optimistically resolve conflicts. Rather than invariantly stalling a transaction when a conflict arises, a less conservative approach to eager conflict management could assume that the conflict will not affect the course of the program, and speculatively let the transaction continue its execution past the conflict. In other words, the pessimistic (early) detection is followed by an optimistic resolution in which actions are deferred until later, so that measures (stalls, aborts) are only taken if correctness is threatened. In this work we focus on write-after-read (WAR) conflicts – situations where a transaction needs to write data that has been read by other concurrent transaction(s) –, though this idea can be applied to other conflicting scenarios as well. WAR conflicts can be resolved by pre-establishing an appropriate commit order, guaranteeing that a consistent view of the memory will be maintained at all times. Figures 1 (c) and (f) introduce this hybrid, *pseudo-optimistic* conflict management scheme, in comparison to the classical pessimistic (eager) and optimistic (lazy) policies. We observe in Figure 1 (c) how the speculation manager must delay the commit of transaction T1 until after T2's commit, or else it would violate the transactional consistency model. In Figure 1 (d) to (f) we show how a similar scenario in which transaction T2 takes longer to complete, and how in this case our speculation-based proposal provides a stall-free resolution strategy that emulates the behaviour of a lazy scheme (an optimal solution for this particular case).

As previously proposed eager HTMs, our system relies on the coherence protocol to detect conflicts as soon as they appear, but unlike them, it attempts to resolve most conflicts without stalling the conflicting transaction, delaying the stalls, if needed, until commit time. With very modest hardware requirements, this pseudo-optimistic policy has the potential of enhancing concurrency among conflicting transactions in a totally transparent manner to the upper levels. First, it enables transparent read-write sharing, allowing a writer to speculatively overwrite old data in the presence of multiple running readers, thus extracting more parallelism and improving performance. Second, by delaying the decision of aborting a transaction as long as possible, our mechanism also acts as an accurate prefetching engine, capturing an advantage of lazy conflict management. Third, it alleviates the negative effects of *starving writer* [3], a performance pathology suffered by eager HTMs like LogTM-SE [21]. Finally, since some cyclic conflicts are now resolved without the need to abort any transaction, the number of aborts is reduced thanks to our proposal, and so is network traffic, which will translate into lower power consumption. In summary, simulation results using GEMS [13] and the STAMP benchmark suite [4] show that our proposal obtains average reductions in execution time and network traffic of 8% and 17%, respectively, when compared to the eager system, while the improvement is more pronounced in terms of execution time (17% reduction) for the lazy one, with virtually the same amount of network traffic.

The rest of the paper is organized as follows: Section 2 covers the related work and motivates the need for implementing more efficient conflict management policies in hardware. In Section 3 we describe in detail our mechanism for the speculative resolution of WAR conflicts. Section 4 evaluates the performance of our proposal, comparing it to classical eager and lazy schemes. We end with Section 5, that summarizes the main conclusions of this study and presents our future work.

2. Motivation and Related Work

Although the mechanisms of conflict and version management can be entirely implemented in software [8], [12], [6], moving some basic transactional functionality to the hardware level is essential to mitigate the performance overheads of the software. Hardware transactional memory systems (HTMs) implement all the required hardware to provide TM semantics [9], [7], [1], [16], [15], [21], [2], while hybrid approaches (HyTMs) push into silicon only

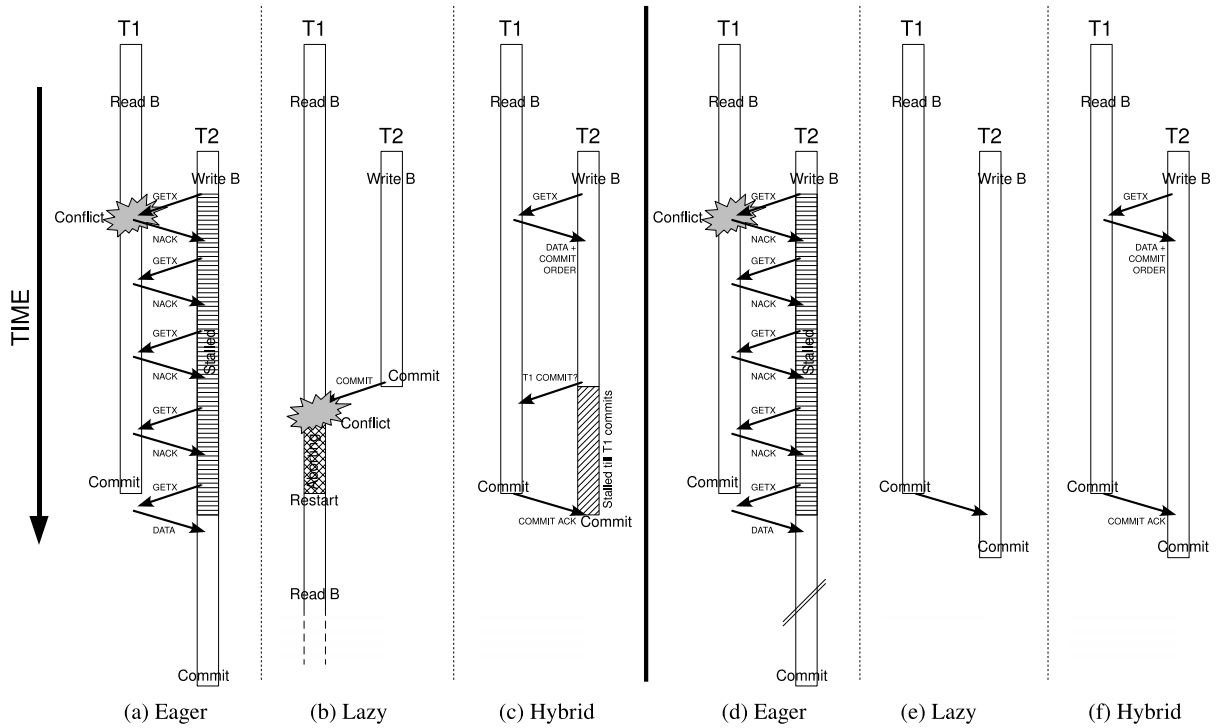

Figure 1. Read-Write conflict management under traditional eager and lazy policies, compared to our proposed hybrid approach.

certain functionalities, so as to accelerate those operations in the software TM (STM) runtime that are critical for performance [10], [14], [17]. Despite the great interest in TM shown by chip manufacturers, they remain prudent about committing to hardware changes and only one major vendor has dared to introduce the first TM-like features in its architecture [19].

HTM systems proposed to date handle fairly efficiently those TM applications composed by small, short running transactions that do not overflow hardware resources and expose few conflicts [15]. Efforts have been made in reducing rollback overhead after abort [20] as well as in virtualization, so as to support unbounded transactions that overflow their private fixed-size hardware structures, survive context switches, page migration, etc. [1], [16], [2]. Reducing the complexity and overheads of such virtualization actions has been a main focus in both HTM and STM, yet little attention has been drawn to the management of conflicts among large transactions from a hardware perspective.

The challenge of large conflicting transactions is addressed by Bobba et al. in TokenTM [2], a proposal that introduces a novel token-based conflict detection scheme adapted from token coherence. TokenTM attempts to accommodate concurrent large transactions and execute them without penalty to non conflicting transactions, introducing minimal overhead to small, short-running transactions.

In FlexTM [17], Shriraman et al. adopt a hybrid approach based on a combination of decoupled hardware primitives controlled by software, which enables policy flexibility in conflict management. FlexTM leverages eager detection, yet allows the software to lazily resolve the conflict (*lazy coherence*), allowing it to exploit incoherence when desired. Like our proposal, FlexTM's *programmable data isolation* enables read-write sharing amongst transactions but it requires additional states to the coherence protocol. The mechanism described in this paper, however, is completely transparent to the software and maintains a coherent cache hierarchy at all times, slightly extending the type and payload of some coherence messages.

HTMs often deal with conflicts in ways that sacrifice concurrency. On the one hand, eager systems that monitor the coherence traffic in order to detect conflicts [9], [1], [16], [15], [21] are able to resolve some of them by stalling the conflicting request until the conflicted transaction releases isolation over its transactionally accessed blocks. Transactions are only aborted in those cases when cyclic conflicts arise, creating a risk of deadlock. This policy conservatively limits parallelism because it prevents a transaction from making progress past the conflicting point, even if the subsequent code operates on data than is not part of any other transaction's read-and-write sets. Figure 2 illustrates two scenarios, where T1 and T3 are independent and T2's read

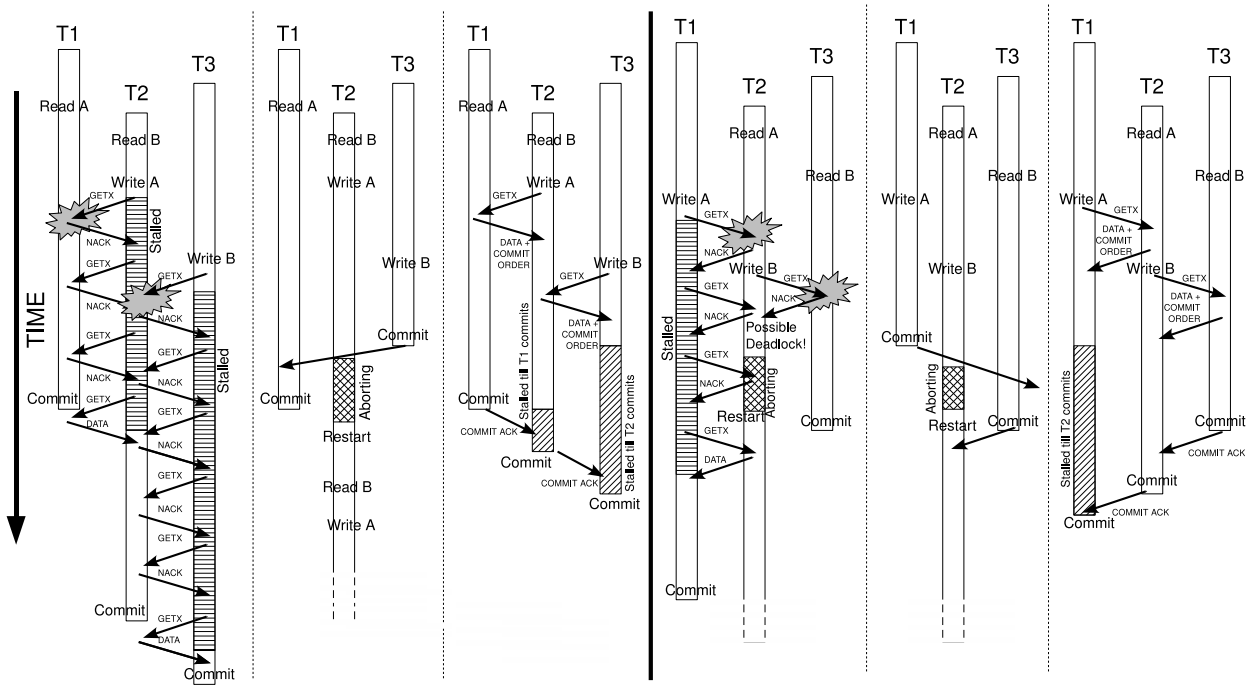

Figure 2. Two chained conflicting scenarios under eager, lazy and hybrid policies.

and write sets overlap with the other two. Figure 2 (a) shows how an eager policy tends to serialize transactions one after another, creating a "chain of conflicts". This serialization effect can be specially harmful when the transactions are long. In other situations, stalling the transaction does not resolve the conflict with a transaction that performs useful work, a pathology called *futile stall* [3] depicted in Figure 2 (d): T1 stalls for T2, but the latter has to abort after all, due to another conflict that creates a potential deadlock[1].

On the other hand, the optimistic conflict detection of lazy HTMs [7], [5] does not favour abundant coarse-grain transactions because i) the larger a transaction's footprint (read&write set) is, the higher is the probability of conflicting with other transactions, and ii) the longer it runs, the higher are the odds that other committing transaction will make it abort. Figures 2 (b) and (e) show how a lazy system would deal with the same two scenarios presented above. In the figure we see that where an eager approach leads to serialization (a) or futile stall (d), the lazy scheme allows both T1 and T3 to commit – since they are independent – at the cost of discarding a fair amount of useful computation performed by T2. However, a lazy conflict management may *starve the elder* [3] when both small and large concurrent transactions access the same data, since the small, short-running transactions often lead the longer ones to abort.

1. We assume a timestamp-based resolution method that prioritizes the older, where age is $T1 > T3 > T2$

Our hybrid conflict management scheme represents a first step towards the integration of both eager and lazy schemes into a single design that attempts to extract their respective virtues without actually having to implement both policies in silicon. The underlaying idea is to take advantage of the coherence protocol for detecting conflicts pessimistically (as soon as possible), yet resolve them optimistically (delaying the stalls as late as possible), in an attempt to extract more parallelism from transactional workloads with abundant conflicts. Figures 2 (c) and (f) describe the behaviour of our scheme in comparison to the classical policies. We can observe how our hybrid approach outperforms them in both presented scenarios, resolving the conflicts without aborts.

3. Speculation-Based Resolution of Conflicts in HTM

The hybrid, pseudo-optimistic conflict management strategy we describe in this section combines a pessimistic (eager) approach to conflict detection with an optimistic conflict resolution strategy based on speculation. As in any eager HTM, conflicts are detected as soon as they occur by monitoring the coherence traffic for transactionally accessed lines. Therefore, our proposal inherits most of the features of systems like LogTM, in regards to conflict detection. For discussion purposes, we consider that transactional accesses are summarized in hash signatures that track both read

and write (R&W) sets [5]. Nonetheless, the speculative resolution presented here is independent from both the transactional bookkeeping scheme as well as the version management policy, and thus it is applicable to any HTM with eager conflict detection. We borrow from LogTM [15] the sticky coherence technique in order to maintain isolation over transactional blocks that are evicted from private cache levels. In summary, in the following elaboration we assume that our mechanism is implemented as an extension to the LogTM-SE system [21]. For that reason, we sometimes refer to LogTM's conflict detection and resolution policies, in order to compare it with our proposal. The chosen environment for this elaboration is a tiled CMP with one level of private cache per core and a physically distributed, logically shared second-level cache that uses a directory protocol to maintain coherency across an unordered network.

3.1. Enabling Read-Write Sharing among Transactions

Eager HTMs monitor the coherence traffic for transactional addresses in order to detect conflicts. When a core attempts to write a data word that has been read by others, it first needs to acquire exclusive ownership over the cache block and hence it must invalidate any copies that may reside in other core's private caches at that moment. Every time a sharer receives an invalidation, it checks the line address through its R&W signatures, to find out whether the address has been accessed by the current transaction. If either signature signals a positive, the sharer responds with a negative acknowledgment indicating that a conflict has just happened. The resolution of the conflict varies depending on the policy. For example, the LogTM system retries the write request repeatedly, until all the transactional readers have committed or aborted, and the writer obtains write permissions over the block.

If the address only belongs to its read signature, a transactional reader is refusing to give exclusive permissions to the writer only because of the constraints imposed by the transactional semantics. The consistency model dictates that a block read during such critical section must conserve the same value when the transaction commits, so that the transaction appears as a *mega-instruction* that was executed at once. However, when the data is no longer needed by the readers –for example, because it was one of the first pointers in a linked data structure that it is traversing–, resolving the conflict by stalling the writer puts unnecessary limits to concurrency.

A different, more optimistic strategy of dealing with this kind of WAR conflicts is to let the writer speculatively obtain the block and modify it –as if no conflict had happenend– while making sure that i) the conflicting readers commit (or abort) before the writer can commit, and ii) the readers only observe the old version of data during the rest of their execution, if needed. To keep the memory hierarchy coherent at all times, readers invalidate their copies of the block as usual and respond with *speculative* invalidation acknowledgements to the requesting writer. On the writer side, the old data is buffered and the *previous readers* are tracked, for the *commit logic* to determine when the writer has permission to commit. On the reader side, each conflicting reader adds the speculative writer to its *next writers* list, which contains the transactions that must be notified when the current reader commits or aborts. Figure 3 depicts the hardware extensions (in grey) required to carry out this hybrid, speculation-based resolution of WAR conflicts, described in this section.

3.1.1. Old Version Buffer. The writer, which naturally backs-up the old version of the block in its log, is responsible for detecting those forwarded requests that come from previous readers (PRs), referred to blocks that have been obtained speculatively. Thus, the writer needs to keep a record of those line addresses, as well as the old copy of the block. In this way, if a forwarded read request for a speculated block comes from a PR, the writer can detect it, locate the old version and respond with an *UncacheableData* message that contains the old data demanded but does not allow the requestor to cache it, to maintain a coherent memory hierarchy at all times. To this end, the writer uses a small, fully associative buffer called *Old Version Buffer* (OVB) to keep a record of the WAR-speculated addresses as well as the old data and the readers associated to it. A simple logical-or of the *Readers* field from all valid OVB entries allows the writer to build its list of *previous readers*, which is used by the *commit logic* to enforce a transactionally consistent commit order. Every reader in this list must have committed or aborted before the writer can commit, so that all the readers appear to have completed before the write instruction executed. Besides saving the old data block in the undo-log, the OVB keeps a copy of it that can be easily located if a read request from a PR is received. This is done in order to keep the logging hardware simple, since the log is usually traversed by software handlers and not in hardware.

3.1.2. Upgrade and reverse conflicts. It is possible that after a WAR conflicting block has been speculatively written, a PR becomes a writer, creating an *upgrade conflict*. Other problematic situations that appear after solving a conflict speculatively are *reverse conflicts*, this is, WAR conflicts on non-speculated blocks that have been read by an speculative writer and then need to be written by some PR (direct cyclic conflicts). Both upgrade and reverse conflicts can only be resolved by aborting one of the two transactions. In order to favour forward progress, a speculative writer is allowed to abort an older PR, as long as the writer has already reached commit and is simply awaiting the readers' commit/abort. When both transactions are running, our scheme simply falls

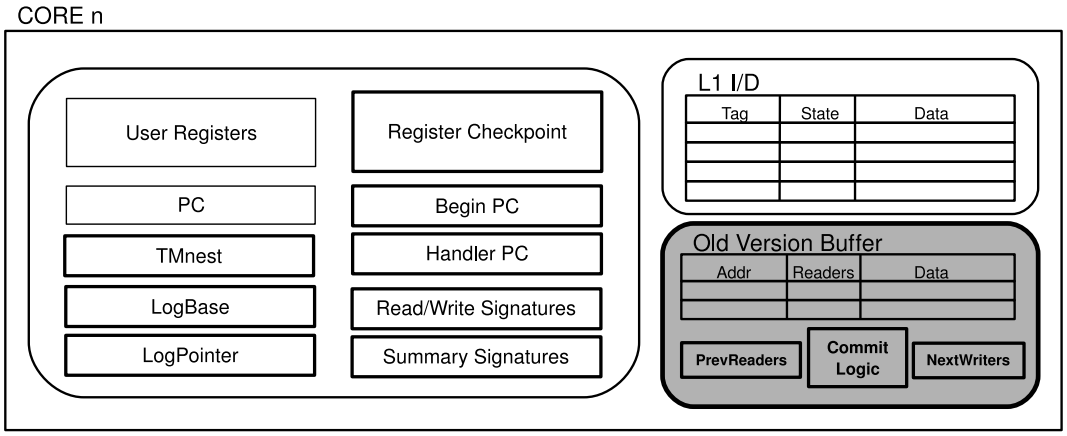

Figure 3. Hardware requirements for our proposed hybrid conflict resolution scheme.

back to the baseline conflict resolution of LogTM (compare timestamps and abort the older). Because the *committing wins* policy could lead to starvation, a transaction that has been explicitly aborted by a younger speculative writer sets its *Disallow Speculation* bit, so that it does not invite any other transaction to start speculation during its re-execution. Because the transaction will not become a PR, it will only be aborted if the baseline timestamp-based resolution scheme dictates it, thus guaranteeing forward progress (it will eventually become the oldest transaction across the system and will have priority over the rest in case of conflict).

3.1.3. Propagation of *Previous Readers*. Cyclic conflicts could still create unfeasible commit orderings if writers invariantly applied speculation. Let us consider this example: T_1 becomes T_2's PR after speculating past a WAR conflict, and then T_2 becomes T_3's PR after another conflict. In this situation, if T_1 wants to write a block that has been read by T_3, applying speculation would create an impossible commit sequence. Therefore, it is necessary that each speculative writer has information not only about its *direct* PRs – given by the field *Readers* of the OVB entries– but also about its *indirect* PRs, which are recursively defined as the direct and indirect PRs of a writer's PRs. Indirect PRs are kept in the *PreviousReaders* register, separated from the direct PRs. Both types of PRs are logically-or'ed into a single bit-vector and sent as an extra field in every "speculative invalidation acknowledgement" message. In this way, the information about a transaction's PRs is propagated to its next writers and so on, allowing conflicting writers to decide whether speculation is applicable. In the example, T_3 will observe that T_1 is one of its indirect PRs, and it will not allow T_1 to speculate.

3.1.4. Recovery from speculation failures. In those cases where a speculative writer has to abort, the directory information for the WAR-speculated blocks needs to be updated so that the original conflicting readers go back to receiving coherence traffic –and detecting conflicts– for those blocks. For each entry in the OVB, a special *PUT sharers* message is issued down to the directory in order to set the "broadcast" mode. Such functionality is used to rebuild the sharers list by broadcasting read/write signature checks to all cores, and it is already provided by the baseline LogTM system (for maintaining isolation over transactional blocks that overflow the directory level). This kind of writeback process must be carried out after the log has been walked and the data blocks restored, and right before the signatures are cleared, so that those blocks remain isolated until the directory is updated. Because the number of entries in the OVB is usually small and the blocks are most likely cached –they were just accessed by the rollback handler–, this extra step does not add much overhead compared to the time spent by the software handler undoing the effects of the transaction.

3.1.5. Commit and abort reports. When a transaction reaches the end of its execution, it checks its OVB looking for previous writers. The transaction has permission to commit immediately as long as the OVB is empty, since this means that there are no PRs and thus there is no risk of violating transactional consistency. However, if it finds valid entries in the OVB, the transaction must wait until all entries have been invalidated. Entries are deallocated when the core has received *committed* or *aborted* report messages for all *Readers* in that entry. One way or the other, once commit permission has been granted, the transaction must send committed reports to the appropriate nodes, if any, as dictated by its *NextWriters* register. The same applies when the transaction is forced to abort: after the log has been restored and the signatures are cleared (isolation is released), an aborted message must be sent to every next writer. Finally, both *PreviousReaders* and *NextWriters* registers are flush-cleared when the core leaves the execution in TM mode.

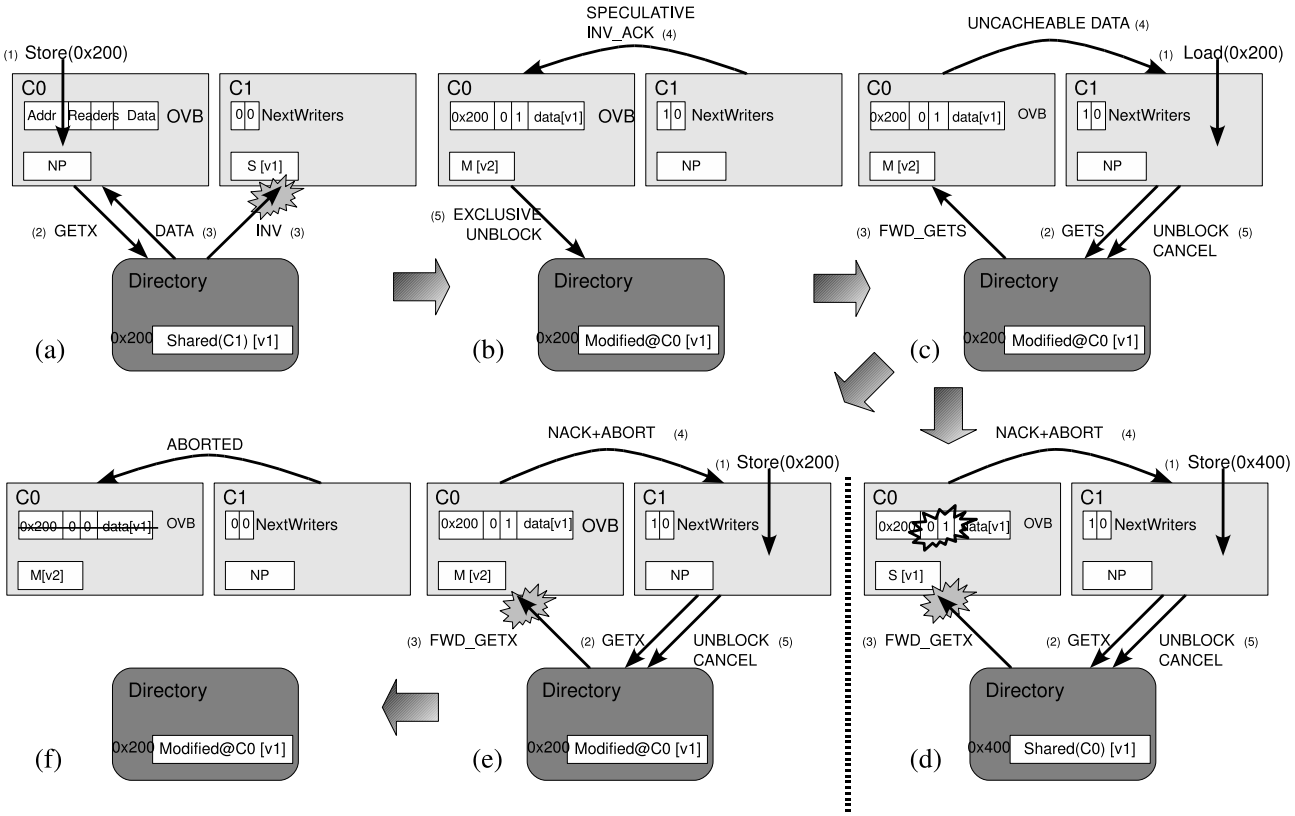

Figure 4. Operation of our hybrid conflict resolution scheme in the presence of WAR conflicts.

3.1.6. OVB overflow. If all OVB entries are busy when a writer receives an invitation, it cannot speculatively resolve the conflict until some direct PR finishes, i.e., some OVB entry is freed upon a commit/abort report message. In this case, the cache controller simply rejects the invitation by considering a negative acknowledgment, and the conflict is resolved by the baseline mechanism (retrying the request). In this scenario, those readers whose invitations got discarded may have imprecise information in their *NextWriters* register, if the writer has not accepted an earlier invitations from those transactions. Hence, spurious commit/abort reports may be received by the writer, though these messages have no effect since they find no OVB entries associated.

3.1.7. Deadlock avoidance at commit-time. Because a committing writer rejects other transaction's requests for its R&W blocks until it receives permission to commit, there is a risk of deadlock that arises when the committer indirectly impedes the forward progress of its own previous readers. This can be avoided by giving higher priority to those *critical* PRs, which some committing speculative writer is waiting for. Thus, committing writers broadcast their direct PRs, so that all other transactions can detect whether they are "nacking" a critical PR and proceed to abort.

3.2. Operation

Figure 4 presents a simple example of how the different hardware components cooperate in order to accomplish the speculative resolution of WAR conflicts in our proposed hybrid conflict management scheme. This example presents two cores, each of them executing a transaction. Initially, Core 1 (C_1) has transactionally read the block with address 0x200, which remains cached in its private level. At this point, Core 0 (C_0) executes a store that needs to write block 0x200, it misses in the local cache and issues a coherence request to obtain exclusive data. The directory responds to C_0 with the data in cache (version 1) and forwards an invalidation request to C_1, whose read signature signals a conflict, as shown in Figure 4 (a). Under the conservative approach of LogTM-SE, C_1 would respond to C_0 with a negative acknowledgement (nack), indicating the conflict and forcing C0 to stall.

As opposed to LogTM-SE', the conflict resolution scheme we propose attempts to solve the conflict optimistically, as depicted in Figure 4 (b). C_1 still detects the conflict, yet it responds with a speculative invalidation acknowledgement (ack) and adds C_1 to its *NextWriters* register. Upon arrival of the ack message, C_0 observes the Speculative bit set and it allocates an entry in the OVB for the old block –obtained

from L1 before the write completes–, setting the bit that corresponds to C_1 in *Readers*. After the OVB entry is ready, the store updates the data block in L1 (v2), and C_0 continues its execution as if no conflict had happened. If C_0 reaches end of the transaction, the commit ordering logic will obtain a bit-vector of previous readers by logically OR-ing the *Readers* field of all OVB valid entries, and will determine whether C_0 can commit immediately or it must wait for the commit/abort report messages from the remaining previous reader transactions.

It is possible that after the block has been speculatively given to a writer, a PR needs to read the block. The miss looks no different than other and is treated as usual (4(c)). If the writer's signatures signal a positive, all valid OVB addresses are compared in parallel to the requested address, to find out whether this request was for an speculated block. If so, the writer responds with the old version (v1) and sets the appropriate bit in the *Readers*, if not already set. The response data message has the *UncacheableData* bit set, forcing the reader to invalidate the block immediately after the load is served. If there was no OVB address match, a conflict is signaled and the response is a regular nack message.

In Figure 4 (d) we can observe a different situation that could happen if C_1 attempts to write a different memory location (0x400) that has been read by the transaction running in C_0, creating a "reverse" WAR conflict between the same two cores. For this reason, a core must always check the current commit ordering before resolving a new conflict speculatively. In the example, C_0 checks its list of PRs and finds that C_1 is already there, so the conflict cannot be solved –it would create an infeasible commit ordering– and one of the transactions should abort. For this example, we assume C_1 has a younger timestamp –contained in the forwarded request– which C_0 compares against its own, and decides to send a explicit abort message to C_0, piggybacked as a bit in the nack response to the conflicting write request. Figure 4 (e) shows an example of a similar situation, but this time the request is for the previously speculated block (*upgrade conflict*). In this case, a write-write conflict is detected by C_0 and resolved in the usual way, without even checking for a OVB match. Considering C_0 older than C_1, the response is a nack plus abort message, similar to the (d) case. Finally, in (f) we observe how the PR clears its *NextWriters* register once it has aborted, and reports back to every writer. Upon arrival of an "aborted" response from core N, the speculative writer clears the N-th bit in all OVB entries and it invalidates those ones that have no remaining bits set, making room for further speculation. When there are no active OVB entries left –all PRs have committed/aborted– , the writer is allowed to commit without further checks. The writer transaction may still be running, or awaiting at its commit point for such permission, ready to trigger the appropriate commit actions.

4. Evaluation

In this section, we evaluate the performance of the proposed conflict management scheme (hybrid). We compare the results with the two systems that use classical conflict management policies: A purely eager HTM such as LogTM-SE [21], and a purely lazy HTM that uses both lazy conflict management and lazy versioning, much like TCC[7].

4.1. Simulation Environment

We use a full-system execution-driven simulation based on the Wisconsin GEMS toolset [13], in conjunction with Virtutech Simics [11]. We use an implementation of the LogTM-SE protocol and the detailed timing model for the memory subsystem of GEMS v2.1, with the Simics in-order processor model. Simics provides functional correctness for the SPARC ISA and boots an unmodified Solaris 10.

We perform our experiments on a tiled CMP system, as described in Table 1. We use a 16-core configuration with private L1 I& D caches and a shared, multibanked L2 cache consisting of 16 banks of 512KB each. The L1 caches maintain inclusion with the L2. The cores and L2 cache banks are connected through a 2D mesh network. The private L1 data caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains a bit vector of sharers and implements the MESI protocol. The size of the *Old Version Buffer* used by our design is 8 entries of 74 bytes each (address, *Readers* and data).

4.2. Workloads

For the evaluation, we use eight transactional benchmarks extracted from the STAMP suite [4] on its version 0.9.10, as described in Table 2. These benchmarks use coarse-grain transactions to execute concurrent tasks on irregular data structures such us graphs or trees. We have also included an earlier version of the benchmark vacation, in which the customer table is not prepopulated during the initialization phase, since this later improvement significantly changed the benchmark behaviour in terms of conflicts. In regards to memory management, we use the simple thread-local memory allocator included in the STAMP library, which we have conveniently modified in order to provide block-aligned addresses on every *malloc* call. Though this can lead to fragmentation and poor cache utilization – we have checked that this does not happen in our applications –, it eliminates most conflicts caused by false data sharing. We found that this simple change had considerable effects on some benchmarks in terms of conflicts and aborted transactions and, without it, the gains of our scheme were even more pronounced for some of the applications.

Table 1. 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 2-way, 1-cycle latency
Old Version Buffer	Private, 8 entries of 74-bytes Fully assoc., 1-cycle latency
L2 cache	Shared, 8MB, unified 4-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh (4x4)
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

Table 2. Workloads and inputs.

Benchmark	Input
BAYES	32 vars, 1K records, 20%p.parent
GENOME	64K segments,1K genes ,seg.length 32
INTRUDER	10%attacks,4 pkts/stream, seed 1
KMEANS	16/16 clusters, t 2048 16-D points
SSCA2	scale 13,clq 1,unid. 1,path 3,edgs 3
VACATION 0.9.6	64K rel,16K tsk,8 qrs/tsk,10%rel,80%usr
VACATION	16K rel,16K tsk,4 qrs/tsk,60%rel,90%usr
YADA	angle 20, input file 633.2

4.3. HTM Systems

We compare our speculation-based scheme against two generic HTM systems implemented in the GEMS simulator, which Bobba *et al.* refer to as EE and LL_b [3]. The eager system is basically a LogTM-SE system that uses a per-thread log in cacheable virtual memory, extends a directory protocol in order to perform conflict detection and encodes R&W sets using hash signatures. LogTM attempts to resolve conflicts through stalls, and uses a deadlock avoidance algorithm based on timestamps: A processor sets a bit if it nacks an older transaction; if in turn it receives a nack from an older transaction, this represents a potential cycle and the transaction aborts. The abort traps to a software handler, which walks the transaction log and restores the old values into memory. As for the lazy system, it employs an infinite write buffer (no overflows) and a zero-cycle commit token bus which helps to compensate for the increased commit latency of the modelled directory-based system. A transaction must acquire this token in order to enter the commit phase, in which it issues exclusive coherence requests for all its written blocks, invalidating (aborting) any transactional readers. Finally, our hybrid system inherits most features from the eager system, except for those regarding the conflict management mechanism presented in the previous section. We use *perfect* signatures – address lists – for all simulations instead of real hash signatures, to isolate the influence of our design from the interference of conflicts due to false positives. All simulated systems employ randomized linear backoff to reduce contention after abort.

4.4. Results

Table 3 summarizes some of the most significative results of the proposed speculation-based conflict resolution mechanism. Column *SpecAttmpts* shows how our scheme achieves over 90% of successful attempts –speculatively written blocks that did not have to be transferred back to the readers' isolation– for the majority of benchmarks. The right half of Table 3 shows the number of commits and aborts that correspond to speculative writer and PRs transactions. More detailed statistics about the type of abort –*upgrade/reverse* conflicts, respectively– are shown in brackets. Note that not all aborts fall into either of these two types. In columns 5 to 7 we present the number of aborted transactions for all three evaluated systems. Our hybrid proposal manages to lower this number to levels that are in some cases well under the other two systems (bayes, genome, both versions of vacation). This partly explains the reductions in execution time and network traffic shown in Figures 5 and 6, respectively. Observing these two figures we see how the hybrid scheme (middle bar, *hybrid* label) performs more regularly than the other two across a range of workloads with different characteristics. When compared to the eager system (left bar, *eager* label), traffic is reduced by 15% on average, while execution time has an 8% average reduction. In contrast, the average performance gain of our proposal is as high as 17% with respect to the lazy system (right bar, lazy label), with roughly the same levels of traffic.

The current, "prepopulated" version of vacation experiences a formidable speedup of almost 2 compared to the eager system, mainly due to the elimination of the starving writer pathology –an inherent advantage of the hybrid conflict management–, reflected on reduced final barrier time of the *hybrid* plot (no thread left behind). vacation 0.9.6 has a slightly different behaviour, as transactions find an empty client table and generate a higher number of insertions in the data structure (see *Wset* in Table 3, column 3). This creates considerable more conflicts, which our hybrid design can tolerate in a way the eager system cannot. In this case the reduction in execution time is mostly due to a lesser number of stalled and backoff cycles, a result of increased concurrency and fewer aborted transactions, respectively. The lazy system performs as good as the hybrid in terms of performance and traffic, for both versions: Since most transactions are independent, they can execute concurrently and commit without the serialization effect of chained conflicts suffered by the eager policy.

For both bayes and yada, the mixture of long and short

Table 3. Committed and aborted transactions. Speculation summary.

	Rset	Wset	Commit	Aborts Eager	Aborts Hybrid	Aborts Lazy	Spec Attempt	% suc cess	writer commit	writer abort	PR commit	PR abort
bayes	161,2	122,4	538	1202	446	714	152	92%	103	10(2/1)	104	187(32/12)
genome	26,9	3,1	40050	2770	916	2039	898	81%	484	213(19/16)	81	628(275/269)
intruder	9,0	3,3	11224	35322	37930	25341	1836	73%	966	661(148/497)	414	10332(330/842)
kmeans	6,2	1,7	8238	6092	5845	1759	23	99%	22	0	0	33(0/0)
ssca2	3	2	47300	221	218	1081	21	100%	21	0	0	21(0/0)
vacation	70,2	8,9	16384	1152	29	587	330	98%	315	3(2/0)	447	24(15/4)
vacat096	91,7	11,6	16384	4581	851	2480	1401	94%	834	61(39/7)	1099	593(111/54)
yada	40,8	24,0	5212	12275	13915	16172	4357	70%	929	1117(211/808)	762	10460(252/2008)

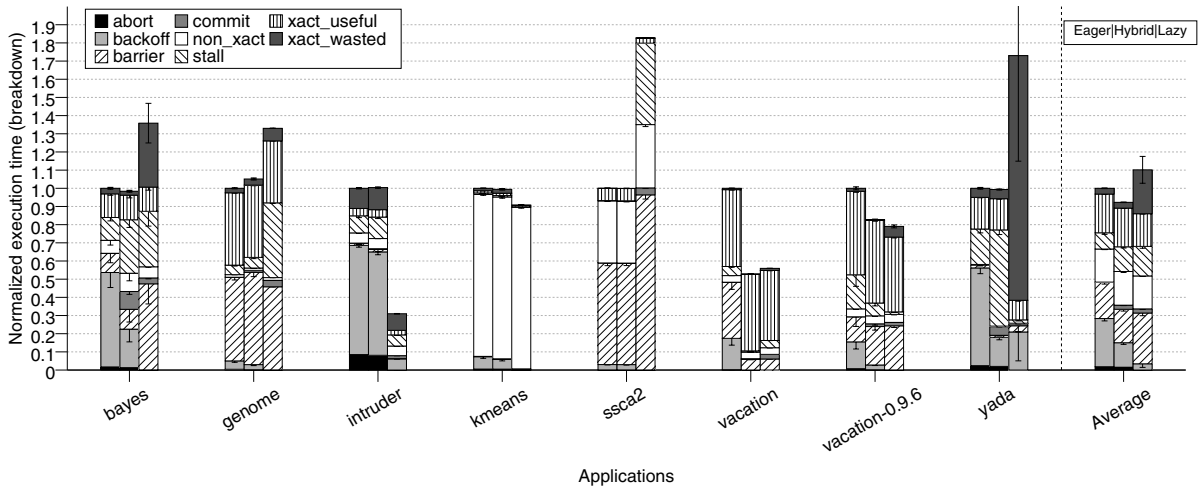

Figure 5. Execution time breakdown.

running transaction with lots of writes makes it a tough scenario for the lazy system to beat the approaches based on eager detection, as the optimistic policy cannot avoid a huge amount of wasted work caused when aborting large transactions (*xact_wasted* portion of the *lazy* plot in Figure 5). This problem is definitely more acute in yada than in bayes, though the lazy transactions in bayes do suffer from a serialized commit effect that does not happen in yada. For the latter benchmark, the hybrid system experiences more aborts than the eager due to the interaction amongst short running speculative writers and long running PRs, which makes the former stall at commit for long periods of time (see *commit* fraction in *hybrid* plot). As a result, the stalled cycles grow because the stalled-at-commit writer retains isolation over blocks that are needed by others. The number of aborts increases in the hybrid system in comparison to the eager, as a result of the priority mechanism used to avoid commit-time deadlocks: Long running critical PRs abort many other transactions that try to reject their high priority requests, though this barely impacts performance because this aborts tend to happen early.

The benchmark genome spends a large portion of its time in transactional execution, but conflicts are frequent only in certain phases of the computation. In those other phases where there are few conflicts, the lazy system has problems of serialized commit that degrade performance. Despite reducing the number of aborts to almost a third, the hybrid system does not perform better than the eager nor does it significantly reduces network traffic. This is due to the abundant refetching of old data by PRs, which requires OVB-to-cache messages whose uncacheable nature increases the latency of reloads.

Intruder is the only benchmark for which the lazy system beats both eager and hybrid systems by far, though most of the reduction comes from the backoff cycles. Both eager and hybrid systems suffer a huge number of aborts, while the lazy policy reduces this number by almost 30%. Thanks to the small transaction footprint (particularly the *Wset*), the lazy approach minimizes the probability of conflict with other concurrent transactions.

The case of ssca2 is straight-forward: very small transactions that barely communicate any data but are concentrated in the same execution phase, resulting in few conflicts but in a huge contention for the commit token. Clearly, this makes

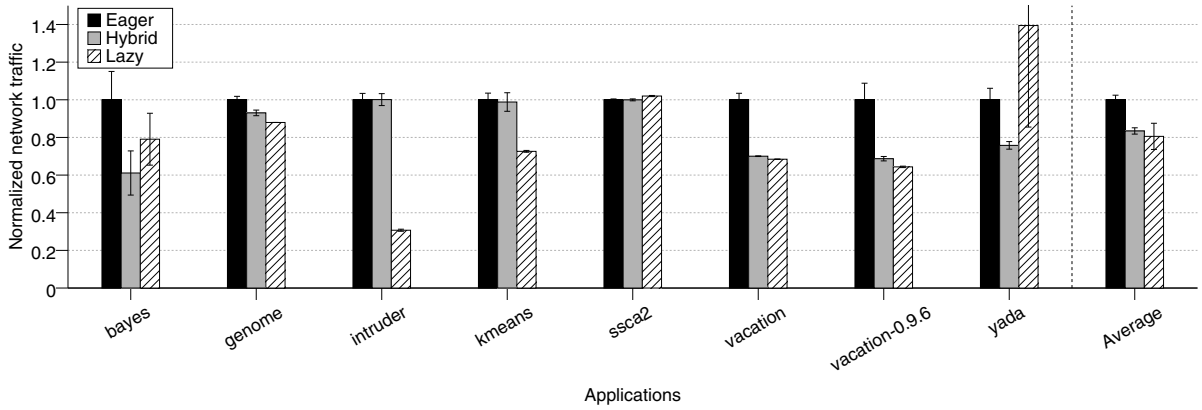

Figure 6. On-chip network traffic.

the eager and hybrid approaches outperform the lazy system by far, because of the forementioned serialized commit pathology.

In kmeans, the fraction of time spent in transactions is so small that there is little difference between all three systems. The lazy case wins because it reduces the amount of aborts and this reflects on the backoff portion of Figure 5 and considerable in network traffic.

5. Conclusions and Future Work

Transactional Memory is not the panacea of the multi-core programming era. It definitely helps programmers to write correct code quicker, yet those who want their multithreaded program to run fast will still need to tune their codes to minimize the amount of transaction conflicts. Considering that few will be willing to make this extra effort, future TM-capable chips are presented with the challenge of extracting thread level parallelism even in unfavorable situations, and their success will partly depend on how well they tolerate programs with abundant conflicts. While the conflict management scheme plays a crucial role in this matter, few are the contributions on HTM systems that focus on the efficient execution of those transactional workloads that are prone to experience conflicts. With this work, we fill that gap by introducing a novel resolution mechanism that applies the idea of speculation to provide a pseudo-optimistic policy. Conceptually laying in between the two classical schemes, our hybrid approach has the ability to combine some of their benefits at a very modest hardware cost. For the selected transactional benchmarks, it outperforms both eager and lazy systems with average reductions in execution time of 8 and 17%, respectively, and it decreases network traffic by another 17% compared to the eager policy.

Furthermore, the idea of early detection and late resolution of conflicts can be extended to scenarios other than the write-after-read case presented in this paper. We are currently extending our design to support speculation past read-after-write conflicts, so as to enhance concurrency of producer-consumer transactions, and our ultimate goal is to present an HTM system that can tackle a wide variety of conflicting situations.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03', as well as by the EU FP7 NoE HiPEAC IST-217068. Rubén Titos is supported by a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152).

References

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb 2005.

[2] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 81–91. Jun 2008.

[3] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 81–91, 2007.

[4] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*. Sept 2008.

[5] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, Jun 2006.

[6] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[7] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, Jun 2004.

[8] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Symposium on Principles of Distributed Computing*, pages 92–101, Jul 2003.

[9] Maurice Herlihy and Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–301, May 1993.

[10] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 209–220, Mar 2006.

[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] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Sep 2005.

[13] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. 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.

[14] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, 2007.

[15] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Feb 2006.

[16] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 494–505, Jun 2005.

[17] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th International Symposium on Computer Architecture*. Jun 2008.

[18] Rubén Titos, Manuel E. Acacio, and José M. García. Directory-based conflict detection in hardware transactional memory. In *Proceedings of the 15th International Conference on High Performance Computing*, pages 541–554, Dec 2008.

[19] Marc Tremblay. Transactional memory for a modern microprocessor. In *Proceedings of the 26th Symposium on Principles of Distributed Computing*, pages 1–1, 2007.

[20] M.M. Waliullah and Per Stenstrröm. Reducing roll-back overhead in transactional memory systems by checkpointing conflicting accesses. In *Proceedings of the 22nd International Parallel & Distributed Processing Symposium*. Apr 2008.

[21] 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 *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, Feb 2007.