Eager beats Lazy: Improving Store Management in Eager Hardware Transactional Memory

Rubén Titos-Gil, *Member, IEEE,* Anurag Negi, *Member, IEEE,* Manuel E. Acacio, *Member, IEEE,* José M. García, *Member, IEEE,* and Per Stenstrom, *Fellow, IEEE*

Abstract—Hardware Transactional Memory (HTM) designs are very sensitive to the manner in which speculative updates from transactions are handled in the system. This study highlights how the lack of effective techniques for store management results in a quick degradation in the performance of eager HTM systems with increasing contention and, thus, lends credence to the belief that eager designs do not perform as well as their lazy counterparts when conflicts abound. In this work we present two simple ways to improve handling of speculative stores – a way to effectively manage lines that exhibit migratory sharing and a way to hide store latency, particularly for those stores that target contended cache lines owned by other concurrent transactions. These two mechanisms yield substantial improvements in execution time when running applications with high contention, allowing eager designs to exceed the performance of lazy ones. Interestingly, the benefits that accrue from these enhancements can be at par with those achieved using more complex system-wide HTM techniques. Coupled with the fact that eager designs are easier to integrate into cache coherent architectures than lazy ones, we claim that with judicious management of stores they represent a more compelling design alternative.

Index Terms—Parallel programming, multicore architectures, transactional memory.

1 Introduction

Hardware Transactional Memory (HTM) [10] is a technique that aims to improve the performance-complexity trade-off involved in parallel programming. It provides conceptually simple atomicity semantics to programmers, whereas the burden of implementing this abstraction in a safe and efficient manner falls on the hardware designer. The industry has recognized the benefits of this approach and the ideas have been implemented in silicon – like IBM BlueGene/Q [18] – or incorporated in architectural specifications – like Intel TSX [4] –, with actual products due to hit the market soon.

Most HTM designs rely upon containment, in some fashion, of updates made in a transaction. Speculative updates can be confined to thread-local structures like private caches [9]. Such versioning of possible future values is termed lazy, where the defining characteristic is that exclusive ownership over speculatively targeted locations is acquired only after a transaction's execution is guaranteed to succeed. Alternatively, updates can be made in-place, which implies early acquisition of written locations, when protocols exist to ensure their isolation and restoration of a consistent state when data-races need resolution. Such a mechanism is termed eager and is utilized by designs like LogTM [15], [25]. A closely related design choice is that of conflict resolution. Eager versioning of updates necessitates eager resolution of conflicts – races must be detected and resolved when an in-place shared memory update is attempted. Lazy versioning, however, allows conflict resolution to be deferred until a transaction tries to commit, thus enabling reader-writer concurrency. This choice of policies has a significant impact on the complexity of HTM designs. Though resolving conflicts *a priori* makes eager systems inherently less efficient at exploiting parallelism, their natural fit onto a cache-coherent CMP substrate makes eager HTM solutions an appealing choice.

Transactions can be limiters of application scalability, making it crucial for an HTM system to execute them fast and with a high commit rate. While eager HTM designs naturally achieve the latter goal in low contention, their performance may suffer in highly contended situations if stores are not efficiently managed. Speeding up transactions by hiding store miss latency becomes important to improve overall concurrency, as it narrows the window of time in which such transactions persist and can cause conflicts. In fact, TM workloads may contain coarse-grained transactions with many more stores than typical fine-grained critical sections associated with locks. To this respect, store buffers are commonly employed to hide the latency involved in performing stores in the cache hierarchy. Transactions present an opportunity where greater freedom in the order in which stores are completed from the store buffer permits optimizations that improve eager HTM performance. Furthermore, certain store misses due to insufficient coherence permissions, i.e. stores that target a line privately cached in shared state (termed *upgrade misses* [1]), can be avoided if cache lines that exhibit migratory sharing patterns [5], [23] (*migratory lines*) are properly managed.

- *R. Titos-Gil, A. Negi and P. Stenstrom are with the Department of Computer Science and Engineering, Chalmers University of Technology, S-41296, Gothenburg, Sweden. E-mail: {ruben.titos, negi, per.stenstrom}@chalmers.se*
- *M. Acacio and J. García are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, E-30100, Murcia, Spain. E-mail: {meacacio,jmgarcia}@ditec.um.es*

Digital Object Indentifier 10.1109/TPDS.2012.315

In this paper, we extend the exploration of transactional store management presented in our previous work [16]. However, our primary focus now is on eager HTM designs. We show that by allowing execution to proceed beyond conflicting stores (by buffering such stores in dedicated structures), performance can be improved in scenarios where eager HTMs have traditionally fared worse than their lazy counterparts. Improving the manner in which lines with strong read-modify-write behavior in transactions are managed also results in major scalability enhancements for applications with small, high-contention transactions. We summarize the novel contributions of this work in the following list:

- An accurate mechanism to dynamically detect and manage migratory lines and prevent aborts in high contention scenarios.
- A novel technique that reduces contention and network traffic by buffering stores to conflicting lines until commit time.
- A commit-timestamp based scheme for prioritization of transactions that are ready to commit.
- An exhaustive evaluation of these techniques against traditional eager HTMs and an idealized lazy HTM design, which does not only show convincing performance improvements but also higher efficiency in the utilisation of common resources.

The rest of this paper is organized as follows. Section 2 presents the motivation for this study. Section 3 explains how simple enhancements can be incorporated into a well-known eager HTM design. Section 4 describes the experimental methodology adopted to evaluate the techniques, and Section 5 presents our results and analyses. Section 6 puts our work here in perspective of other prior work, and Section 7 summarizes the paper.

2 Motivation

Two concurrent transactions conflict when accesses from each target the same memory location and at least one of the accesses is a write. Therefore, effective management of transactional stores by the underlying HTM system is key to effectively extracting available parallelism in TM workloads, particularly of those stores that target contended cache lines. In this section we motivate our study by presenting real examples of how poor management of stores can lead to substantial performance losses.

2.1 Migratory lines

Transactions in several common usage scenarios exhibit read-modify-write (RMW) behavior for certain memory locations. Fine grain critical sections that fetch and update a shared counter or modify a pointer at the head of a data structure are typical examples of this situation. When contention is observed in an application with this kind of transactions, a handful of these *migratory* cache lines are often the source of most conflicts. Transactions in this case simply serve as a mutex that synchronizes accesses to a shared variable, so that once a thread has loaded the value (fetched the cache line), it remains isolated until the thread completes the modify and write steps atomically. Since there is no available parallelism when concurrent transactions access a migratory line, optimistic concurrency is not benefitial as it may lead to negative interference amongst threads. Without any optimization, TM overheads can quickly add up due to unnecessary aborts (of transactions that are able to read the line before they see a write) and delayed processing of requests at the directory, since it must process a much larger number of requests than in the optimized case.

Some compiler optimizations may request for exclusive ownership at read events if a location is assured to be written subsequently [22]. However, it is not possible to do so in all such cases. Moreover, reads and writes to different variables present on the same cache line may escape this optimization. Moreover, contention characteristics may show dependency on the dataset and, therefore, be difficult to ascertain at compile time.

Hardware predictors can be employed to detect migratory lines [11] and avoid the *dueling upgrades* that arise when concurrent transactions read, modify and write the same line. Prior work on HTM [2] attempts to provide a solution to this problem by using write-set predictors, storing a certain number of cache line addresses that have seen read-modify-write behavior. However, our analysis indicates that this mechanism still leaves a lot of room for improvement. Despite write predictors, sudden drops in scalability are seen in benchmarks with short RMW transactions, largely due to a marked increase in the number of aborts. More details about such analysis can be found in Section 2.1 of the supplemental material, which contains a case study for the STAMP benchmark *intruder* [3]. We will show later how appropriate tracking and management of migratory lines significantly improves the scalability of intruder.

2.2 Store misses and conflicting writes

Prior work on eager HTM systems [15], [25] has invariably handled store misses by stalling execution at the core until the write performs in the private cache. Unlike eager systems, lazy designs in the literature [2] hide write miss latency by modeling a private per-processor store buffer. Modeling store management in such opposite ways for eager and lazy HTM may lead to unfair performance comparisons, since in-place versioning does not preclude the use of store buffers to hide miss latencies until writes perform in the cache hierarchy. Their effect can be particularly visible in workloads with large transactions (many stores) or high contention. Our analysis on workloads composed by coarse grain transactions reveals that indeed store misses are responsible for a significant portion of the time spent in active transactional execution. Our analysis in Section 2.2 of the supplemental material quantitatively shows that write misses account for an important fraction of

the transactional cycles in several STAMP benchmarks. Thus, the performance implications of store buffering must not be neglected when comparing against lazy designs, particularly if we consider workloads that spend most time in transactions that update a large number of cache lines, like *yada*.

Conflicting stores in eager systems have also been typically handled by stalling execution at the core while the conflicting condition persists. This can be done via special notifications from a conflicting remote transaction or through repeated retries of the conflicting request. With store buffering, the processor can continue execution past a conflicting update, a feature typically found in lazy HTM designs. The presence of a buffering/forwarding mechanism for conflicting stores makes eager designs capable of sidestepping false write-after-read and write-after-write dependencies and enables more concurrency, improving performance in scenarios where conflicts dissipate without resulting in aborts.

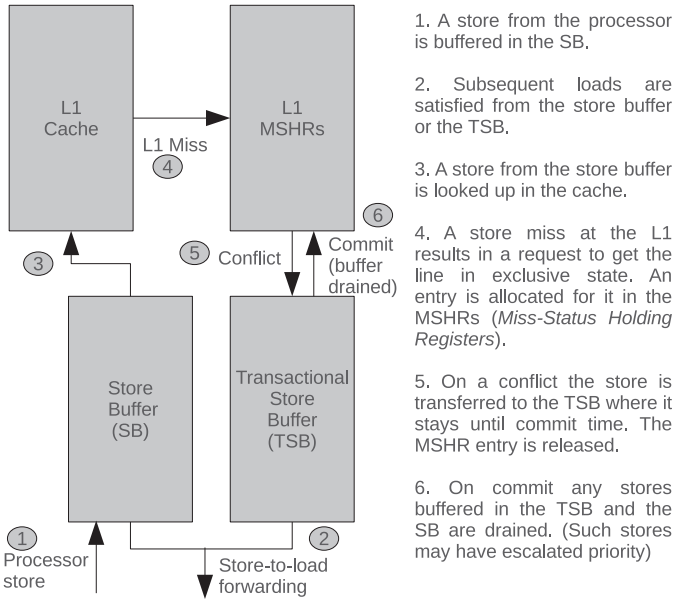

Fig. 1. Buffering transactional stores.

3 Efficient Management of Stores in Eager HTM Systems

As stated earlier, this paper attempts to improve management of transactional stores by targeting two aspects of such accesses – namely, latency and migratory behavior. The design changes required to support each of these are detailed in the following subsections.

3.1 Hiding latency

Structural optimizations such as store buffers are commonly used to hide write miss latencies. In the context of eager HTM systems, it is possible to completely overlap write miss latency as long as the transaction has useful execution to perform on the core. Buffered writes are released in a controlled manner into the private cache, so that in-place updates to shared memory are attempted in a non-blocking manner while the core continues to run ahead. Logging of old values in the undo-log as necessitated by eager versioning is also taken out of the critical path. Thus, transactional updates never stall execution on the core. Write misses to non-contended data only penalize execution if they happen rather close to end of the transaction, in which case the core needs to stall until all pending updates complete.

Strong atomicity and isolation guarantees provided by HTM allow writes emanating from a transaction to complete in any order without affecting the consistent view of memory provided by the parallel architecture. In spite of this opportunity, prior research on eager HTM usually resorts to completing writes in-order, resulting in a stall when a request for exclusive ownership encounters a conflict. However, the presence of a buffering/forwarding mechanism opens up more parallelism to be exploited, since conflicting stores now do not block completion of other memory accesses. One approach consists in retrying conflicting requests in the background while execution of the transaction continues.

Instead of retrying conflicting accesses, we propose a different solution in which such accesses are held back until commit time. The execution of the transaction proceeds with loads to conflicting locations being forwarded data from this buffer. To achieve this we employ a structure called the Transactional Store Buffer (TSB) to buffer conflicting stores seen during the execution of a transaction. The manner in which various components interact is depicted in Figure 1. The TSB, along with the traditional store buffer (SB), is able to satisfy local loads that attempt to read locations targeted by stores contained in the two buffers. Stores from the core are buffered in the SB (step 1,2) until they can be issued into the cache hierarchy for completion (step 3). Store misses (including stores that target cache lines in the shared state) in the first level cache (step 4) are injected into the network for processing at the corresponding L2 directory bank. Entries are moved from SB to TSB when exclusive requests issued by the cache controller upon write misses turn out to be conflicting with one or more concurrent transactions in the system. Remote caches respond with negative acknowledgments (NACKs) to the local controller indicating a conflict. In such a case the store is entered into the TSB (step 5). Such stores are buffered in the TSB until commit time. Stores can be retried if the TSB is full. Section 5 recommends suitable sizing for the SB/TSB pair. On commit, the contents of the TSB are drained (step 6). Commit is deemed complete only after all buffered stores are successful.

Eventually, if the transaction reaches the *txcommit* instruction, buffered conflicting accesses are drained from the TSB into the memory system. These request are given higher priority, thereby favoring transactions that are ready to commit. Coherence requests generated at this point carry a bit indicating its commit status. In this

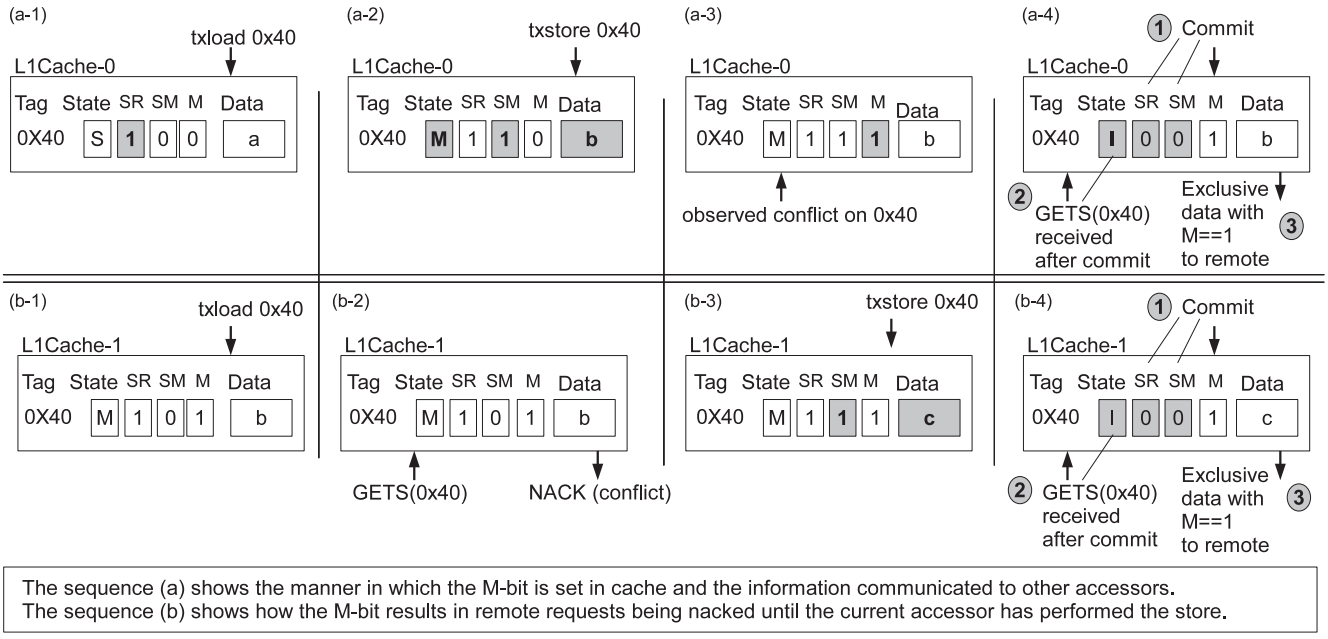

Fig. 2. Handling of migratory lines.

way, conflicts between a committer and a non-committer always favour the former, whereas conflicts between two committers prioritize the transaction that reached commit earlier. This is achieved through a *commit timestamp* that is acquired when the *tx-commit* instruction is reached, and piggybacked on coherence requests instead of the usual *age-based* timestamp. Note that the implementation of this priority scheme leverages all the timestamp-management hardware and registers already present in LogTM, and only requires an additional extra register to maintain the commit timestamp.

3.2 Handling migratory lines

In addition to SR (speculatively read) and SM (speculatively modified) bits for annotating transactionally accessed data, we introduce a new metadata bit called the M-bit for each cache line (M indicates migratory). The bit is set when a conflict is noticed on a line for which both SR and SM bits are set. The state of this bit is conveyed to remote cores when forwarding cache blocks. The bit is cleared using conditional gang-clear operation at transaction commit when either SR or SM is set, and thus remains unchanged when both (or none) of them are active. Non-transactional stores also result in the M bit being reset. We decouple transactional metadata from the cache to permit greater flexibility in its manipulation through conditional clear logic. Modifying SRAM cells in the data cache is non-trivial as the technique requires *gang operations* (operations that are applied en masse for all cache blocks) that compute a logic value for the M-bit based on the state of multiple other metadata bits.

Figure 2 shows how the new cache line annotation works. Shaded boxes highlight changes on each step. The sequence (a) shows how the M-bit is set locally upon noticing a conflict (a-3) on a cache line that has been both read (a-1) and written (a-2) in the transaction. After commit (a-4), the M bit indicates that line must be forwarded to the requesting core in the exclusive state, even for shared (GETS) coherence requests. The sequence (b) then shows how the same line is handled by the new owner (L1Cache-1), after it has obtained exclusive data –along with the piggybacked M-bit– and served its load miss (b-1). We see how remote shared requests to the SR line are now conservatively treated as conflicting and NACKed (b-2). Thus, transactions that are likely to write to a line are given a chance to do so (b-3) with concurrent accesses from other transactions being disallowed. This is expected to have little negative impact on overall concurrency since read-modify-write (RMW) accesses typically offer little parallelism.

Lines detected as migratory may dynamically change its RMW behaviour throughout the execution of the application. It is also possible for a contended line to be mispredicted as migratory, when many readers exist but only some of them intend to write it. Our scheme quickly escapes these mispredictions: After nacking a given number of times the same read request, the owner is forced to relinquish write permissions, clear the M bit and provide a shared copy of the line to the reader. The current retry count for each access is piggybacked in the coherence request, and checked against a threshold by the owner of the migratory line. Low threshold values may cause exclusive ownership to be relinquished before the write to an actual RMW line takes place, while higher values increase the stall penalty of mispredictions. In our experiments, we find that a retry threshold of 4 is a good tradeoff for accurately detecting mispredictions without incurring in a significant penalty for concurrent readers.

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 4-way, 1-cycle latency
Old Value Buffer	8 cache lines
L2 cache	Shared, 512KB per tile, unified 8-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

TABLE 2
HTM configurations evaluated in Section 5.

Eager	
EE_base	LogTM-SE (GEMS baseline)
EE_hwabrt	Always abort in hardware (unlimited OVB)
EE_swabrt	Always trap to software on abort (no OVB)
EE_tsb	Use Transactional Store Buffer (TSB)
EE_migr	Use Migratory Pattern Detection (MIGR)
EE_tsb_migr	TSB + MIGR
Lazy	
LL_ideal	Ideal commits + Ideal buffering
LL_ideal_bufferL1	Ideal commits + Emulate buffering in L1

4 Evaluation Methodology

In this section we describe our evaluation methodology, simulation environment, HTM systems and workloads.

4.1 Experimental Setup

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set [14], in conjunction with Virtutech Simics [13]. We use the detailed timing model for the memory subsystem provided by GEMS, with the Simics in-order processor model. Simics provides functional simulation of the SPARC ISA and boots an unmodified Solaris 10. We perform our experiments on a 16-core tiled CMP system, as described in Table 1. The private 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. In all our experiments we use an ideal book-keeping scheme to track both read and write sets (*perfect signatures*). For each workload-configuration pair we gather average statistics over 10 randomized runs.

We present parallel execution time divided into disjoint components, each one corresponding to sum of the cycles spent by each thread in a given state during its execution. The components are described in the online supplemental material (Section 4.1). Besides comparing execution times, we also look at other metrics to estimate the amount of resources utilized by each design, and by extension, its energy efficiency.

4.2 HTM Systems

Table 2 summarizes the systems and configurations evaluated in Section 5. We have implemented the techniques of store buffering and detection of migratory lines in the context of eager HTM systems. We have also developed an idealized lazy HTM design in order to compare the improvements achieved by the aforementioned techniques against the best performing lazy counterpart. The reader is referred to the supplemental material available online (Section 4.2) for more details about each design.

4.3 Workloads

We have selected seven (out of eight) transactional applications from the STAMP suite [3]. The application *bayes* was excluded since it exhibits unpredictable behaviour and high variability in its execution time [8]. For kmeans and vacation, only results for the high contention input are shown, as these benchmarks exhibit barely no remarkable performance variations between HTM systems for both contention configurations. Recommended input parameters, detailed in [3], were used. Small inputs sizes were used for all workloads. Medium length runs (denoted by '+') were also included for five applications that show widely varying characteristics.

5 Results

5.1 Baseline eager vs. ideal lazy systems

Figure 3 shows the relative performance of the baseline eager (EE) systems available in GEMS, compared to our lazy design with idealized versioning and commit schemes. The two leftmost plots in the figure correspond to two eager configurations which model, respectively, no rollback penalty (*EE_hwabort*) and a fully software-handled log unroll (*EE_swabort*). Execution times are normalized to the *EE_base* system (not shown in this figure), which can only handle aborts with no rollback overhead if the transaction has written less than 8 cache lines. By breaking execution time into disjoint components, we can see how important a role rollback overhead plays on EE performance, as opposed to lazy systems which can easily discard the speculative state in a few cycles by performing conditional gang-invalidation at the private cache level. We observe in Figure 3 that overall eager performance is worse than lazy even when rollback penalty is completely removed from EE systems. This is particularly noticeable in applications with moderate to high levels of contention like genome, intruder or yada. Figure 3 also shows how much performance there is to gain by speeding up aborts on EE systems. We see that the extra overhead of handling aborts always in software (*EE_swabrt*) does not cause a significant slowdown in most applications, with respect to *EE_hwabrt*. Besides, the latter system performs comparably to *EE_base* (reference of the normalization), indicating that an 8-entry OVB is able to contain all the speculative updates in most cases. Even in contended workloads formed by large

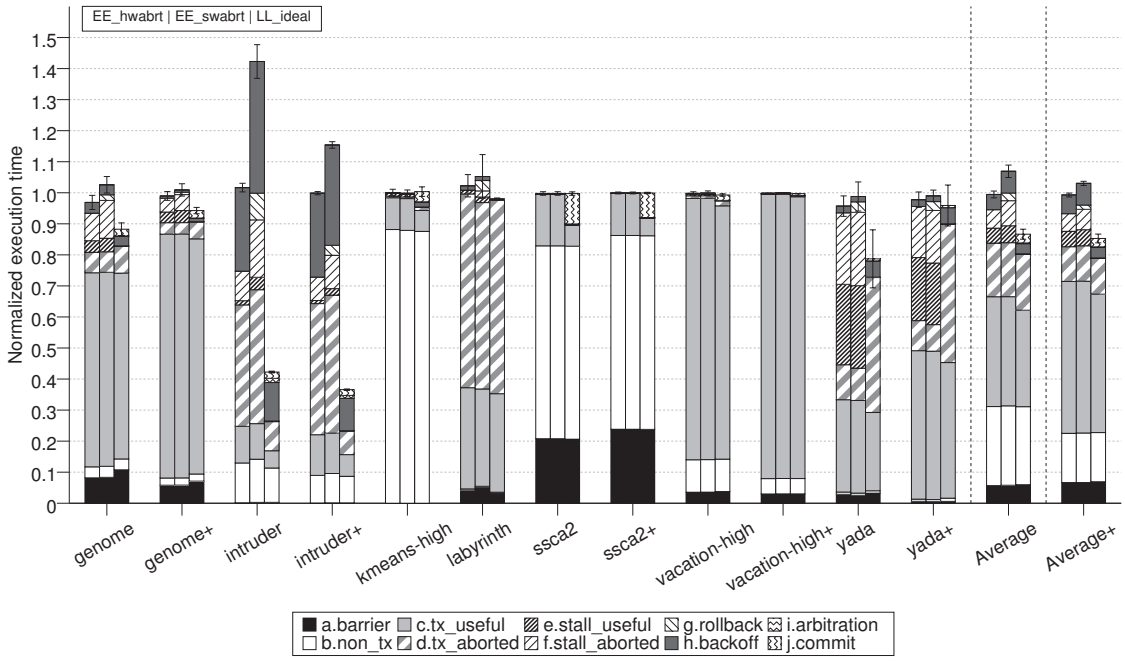

Fig. 3. Relative performance of baseline eager vs. ideal lazy systems.

transactions (like yada or labyrinth), rollback penalty is not that important, as such transaction size in itself leads to a low overall abort count. Only for benchmarks with very high contention over many small transactions (intruder), reducing rollback overhead has a significant impact. In this case, even a small increase in rollback latency translates into even more contention (aborts), since isolation over the contended lines is not released until the software log unroll has completed. For more details about the performance differences between EE and LL systems on each benchmark, we refer the reader to the supplemental material (Section 5.1).

5.2 Migratory pattern detection

Figure 4 compares the performance achieved by each of the two techniques described in Section 3, with respect to our eager baseline. Table 3 shows detailed numbers of commits and aborts for each atomic block – identified by its TID. The table compares the *EE_base* system (columns labelled as *Base*) against *EE_tsb_migr*, which uses both optimizations (columns labelled as *Opt*). The *Diff* column shows the percentage of reduction in the number of aborts achieved by the latter with respect to the former. Bold numbers in this column indicate atomic blocks that exhibit the migratory pattern.

As we see in Table 3, most STAMP benchmarks have at least one transaction that performs read-modify-write (RMW) operations over a shared cached line. This type of transactions often corresponds to the insertion/extraction of work units from a global pool (intruder, labyrinth, yada), though in some other cases it comprises a simple *fetch&add* operation over a global scalar value (kmeans, ssca2). Table 3 quantitatively shows how the optimistic approach to concurrency control is counterproductive for these RMW transactions – there is simply no parallelism available to exploit. By properly managing migratory lines we can eliminate well over 90% of the aborts in most cases. In intruder, frequent RMW accesses to two global queues force *EE_base* to abort a very large number of TID0 and TID2 transactions – indeed much larger than the actual number of commits, which are responsible for the large *txaborted* and *backoff* components seen in Figure 4. In contrast, the *EE_migr* system quickly detects these lines that are read, modified and written in each *queue_pop* operation, serializing accesses and thus avoiding between 80 and 100% of the aborts for TID0 and TID2 seen in the baseline. With this optimization alone, *EE_migr* achieves speedups of 2,5 and 3,3 for intruder and intruder+, respectively, relative to *EE_base*. In the rest of the workloads, RMW transactions have a much smaller weight in the total execution time and thus the optimization does not translate into significant performance gains. In spite of the reduction in the number of aborts, benchmarks like kmeans, ssca2 and yada show only marginal gains. On the other hand, genome and labyrinth present a slight 1-2% slowdown due to longer stalls seen by reader transactions that try to fetch lines wrongly predicted as migratory (i.e. lines that have both M and SR bit sets, but do not become part of the write set for a long time).

5.3 Transactional store buffering

As depicted by the rightmost plots in Figure 4 (*EE_tsb*), transactional store buffering improves application per-

TABLE 3
Average committed and aborted transactions per atomic block, for baseline and optimized eager systems.

	TID0				TID1				TID2				TID3				TID4			
	# Tx	Aborts			# Tx	Aborts			# Tx	Aborts			# Tx	Aborts			# Tx	Aborts		
		Base	Opt	Diff		Base	Opt	Diff		Base	Opt	Diff		Base	Opt	Diff		Base	Opt	Diff
genome	1376	623	598	4,0%	241	0	0	-	3615	357	474	-32,9%	241	2	1	65,2%	449	132	116	11,94%
genome+	2736	538	530	1,4%	481	0	0	-	14911	521	715	-37,2%	481	1	0	66,7%	879	126	130	-3,66%
intruder	3769	14607	10	**99,9%**	3753	7700	2771	64,0%	3753	7425	507	**93,2%**	-	-	-	-	-	-	-	-
intruder+	18322	109174	21	**100,0%**	18306	10849	5872	45,9%	18306	13729	2503	**81,8%**	-	-	-	-	-	-	-	-
kmeans-h	6144	426	15	**96,5%**	2046	171	5	**97,4%**	48	0	0	-	-	-	-	-	-	-	-	-
labyrinth	112	617	2	**99,6%**	96	294	229	22,1%	16	2	0,8	**60,0%**	-	-	-	-	-	-	-	-
ssca2	16	51	5	**89,4%**	16	60	22	**63,7%**	47267	168	185	-10,0%	-	-	-	-	-	-	-	-
ssca2+	16	13	6	**58,6%**	16	51	24	**53,3%**	93695	126	140	-11,3%	-	-	-	-	-	-	-	-
vacation-h	3688	116	103	11,9%	204	0	0	-	204	5,1	6	-17,6%	-	-	-	-	-	-	-	-
vacation-h+	3688	0,3	4	-1233%	204	0	0	-	204	0	0	-	-	-	-	-	-	-	-	-
yada	1344	928	52	**94,4%**	1328	0	0	-	911	2243	1865	16,9%	911	0	0	-	911	335	23	**93,19%**
yada+	3180	1988	118	**94,0%**	3164	0	0	-	2620	3453	2598	24,7%	2620	0	0	-	2620	1055	93	**91,18%**

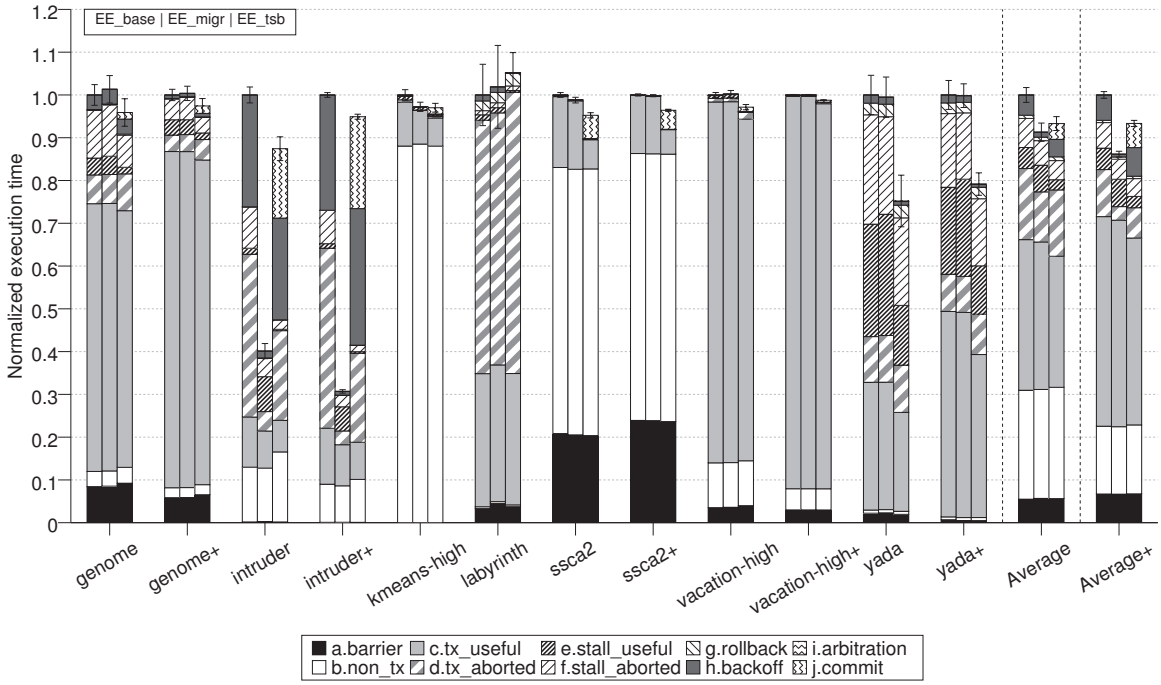

Fig. 4. Relative performance of baseline vs. optimized eager systems.

formance for the vast majority of the benchmarks.

The latency hiding effects of the TSB are most noticeable in workloads with large, contended transactions like yada. The main transaction in yada (TID2) writes over 70 different cache lines on average throughout the course of its execution. There are several reasons behind the 20-25% reduction in the execution time of yada shown in Figure 4. First, overlapping write misses with computation reduces average transaction duration (note the considerable shrink in the *txuseful* component), which in turn narrows the window of contention and therefore decreases the number of TID2 aborts, as shown in Table 3. Second, by allowing transactions to execute past conflicting stores, many stalls due to write-after-read conflicts are avoided (note the reduction in *stall_useful*), further reducing transaction duration. While these false dependencies invariably affect *EE_base*, *EE_tsb* is able to completely sidestep them if the reader transaction ends (commits or aborts) before the writer reaches commit, effectively allowing reader-writer sharing. If the writer reaches commit first, then it forces the reader to abort, since its exclusive requests generated during TSB drainage are prioritized over non-committing transactions, in order to favour forward progress. Nonetheless, *EE_tsb* is capable of exploiting more parallelism than when execution is stalled at the conflicting write.

The ability to run past a contended write may cause aborts in *EE_tsb* for conflicts which *EE_base* can successfully resolve via stalls. We observe in Table 3 how in genome, ssca2 and vacation, the optimized eager system increases the number of aborts of transaction TID2, with respect to the baseline – note that a negative percentage

of reduction in *Diff* represents an increase. In spite of this undesired effect, the benefits of overlapping non-contended store misses with useful transactional execution outweigh the extra aborts seen when conflicts appear. Figure 4 shows overall performance improvements up to 5% for all three benchmarks. In ssca2, store miss latency cannot be completely hidden because the commit instruction in TID2 is often reached before the miss has been serviced. These cycles spent stalled at commit until the TSB is fully drained, accounted for as *commit* cycles in *EE_tsb*, and are clearly visible in ssca2.

Labyrinth is the only benchmark that suffers a 5% performance degradation relative to the baseline when store buffering is introduced. Despite reducing the rollbacks that its main transaction experiences by 22%, the *txaborted* component increases around 6%, while *stall_useful* is shrunk to a barely noticeable fraction in comparison to the baseline. In particular, *EE_tsb* suffers 25% more program-triggered aborts during path validation, which are the most expensive ones since they take place at the very end of a long running transaction. The reason behind this undesired effect is related to the early release of addresses from the read set of TID1 after snapshot of the global grid is taken at the beginning of TID1. With buffering, written cache lines targeted during path validation are not immediately isolated if a conflict happens, increasing the window of time during which other reader transactions in their privatization step can observe *almost stale* data – free grid points about to be marked as busy. Because of early release, it is possible that when the new busy values become globally visible, some readers do not detect a write-read conflict right away and instead continue with the routing phase, only to discover much later that solution found overlaps with another route already committed into the global grid.

TSB Occupancy. Section 5.3 in the supplemental material presents qualitative numbers on the maximum utilization of the TSB. We find that even transactions with large footprints often buffer less than 128 bytes on average.

5.4 Comparison against idealized LL systems

Figure 5 shows the relative performance of the EE configuration that uses both transactional store buffering and migratory pattern detection (*EE_tsb_migr*), against two idealized lazy implementations. Execution times are normalized to *EE_base*, as in previous plots. An analysis of the relative efficiency of both eager and lazy designs, which compares the resource utilisation of the two opposite design alternatives, is available in the online supplemental material (Section 5.4).

Figure 5 reveals several interesting results that break with a common perception in the HTM research community, which has been typically biased against EE designs based on their supposedly lower performance [3], [20]. First, we notice that our optimized eager system tracks and even outperforms an ideal lazy implementation. While EE and LL performance is similar when using small inputs, the improvement of eager over lazy becomes more significant in larger runs that represent a better picture of real-world workloads. Indeed, once these store-related optimizations are in place, the eager variant also outperforms ideal lazy in highly contended benchmarks like intruder and yada, workloads in which lazy policies of versioning and conflict management had typically exhibited clear advantages over eager ones. For a more comprehensive analysis of the performance differences shown in Figure 5, we refer the reader to Section 5.3 of the supplemental material.

Genome. As we see in Figure 5, the improvement over the *EE_base* baseline achieved by the optimized EE system is minimal, for the reasons explained in the previous subsection. Nonetheless, we notice that while the fully ideal lazy system outperforms EE by 5-10%, the differences between EE and LL are less pronounced (1-5%) if we model the effects of using the L1 cache for buffering speculative state (*LL_ideal_bufferL1*).

Intruder. The optimized eager system achieves reductions in execution time of 70-75% over the EE baseline, compared to a more modest 58-64% over the lazy system. Thanks to the more efficient management of stores, the scalability of intruder improves dramatically to reach speedups of over 8,5X for 16 threads, as opposed to the poor 2,6X speedup over single thread runs of the baseline. Most of the advantage of eager over lazy in intruder comes from a significantly lower number of aborts due to RMW transactions, which in turn translates into an impressive reduction in the *backoff* component.

Kmeans, ssca2, vacation. These workloads show less performance variability, though eager still comes out slightly ahead of lazy. For kmeans, backoff overhead is barely visible in EE as a result of a lower number of aborts than LL. Furthermore, in these three benchmarks store miss penalty is partially (ssca) or totally (kmeans, vacation) hidden by the eager system, while in the lazy case write misses are during commit, when it is too late to overlap the miss latency with transactional execution (hence the larger *commit* component in LL systems).

Labyrinth. The optimizations to the eager baseline do not have a positive impact, resulting in a slight performance degradation of under 5% over *EE_base* and 8% with respect to *LL_ideal*, for the reasons described earlier. As we can see in Figure 5, the main reason for the increase in execution time seen in EE is a substantial *rollback* component, which is due to the very large footprint of its highly contended main transaction (TID1). As opposed to EE, lazy systems easily discard speculative state in a few cycles. However, the difference in execution time with *LL_ideal_bufferL1* is less notable, as the number of contamination misses is large [16].

yada. The results for this application are also worth analyzing in more depth. Whereas the EE system only performs slightly better (5%) than LL for the small input, the difference becomes significantly more pronounced (17%) when using medium-sized inputs. As the level of contention affecting its large transactions decreases, the

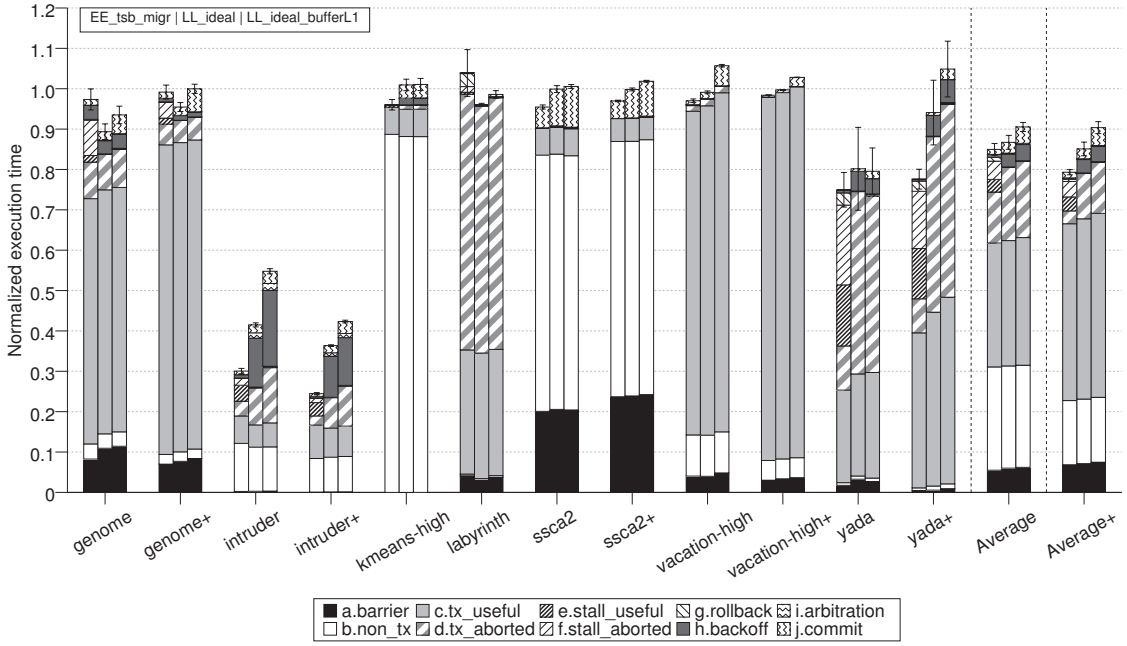

Fig. 5. Relative performance of EE with effective store management vs. two idealized LL configurations.

ability to resolve some read-write conflicts (true dependencies) through stalls rather than aborts becomes more and more advantageous for the eager design. We see in Figure 5 how the *txaborted* component in the EE system is very small compared to the lazy systems, in part as a result of its relatively large fraction of *stall_useful* cycles. As indicated by the *backoff* component, the number of lazy aborts is far larger than in the eager system.

6 Related Work

Concepts first described by Herlihy and Moss [10] have since been leveraged by various HTM policy implementations. Two main categories classify HTM designs: lazy and eager. Lazy designs [9], [17], [24] implement buffering in private caches. Our work in this paper focuses on eager designs which are characterized by in-place modification during speculative stores. The first eager design was introduced by Moore et al. [15]. This was later followed by several variants that improved upon various aspects [12], [25]. However, prior work has not noted the high degree of sensitivity in performance that eager designs exhibit to variations in the management of speculative stores. Small changes in handling updates and conflicting stores can have a large impact on overall performance. More importantly, simple changes in store management can potentially yield greater benefits than using more elaborate schemes proposed in prior studies.

A characterization of STAMP benchmarks [3] showed eager HTM designs perform poorly under high contention, sometimes being worse off than STM implementations. Shriraman et al. [20] performed a comparative study of contention management policies in a hybrid FlexTM [21] based design. This study claims that lazy contention management achieves higher performance than those with eager management. We would like to emphasize that conflict resolution policy is a factor that contributes towards overall HTM system performance but it is not the sole one. Effective management of speculative updates can tip the scales. In particular, simple local optimizations can have a large impact on high-contention performance in eager HTMs.

Sanyal et al. [19] proposed schemes, involving both paging hardware and the operating system, to manage thread-local data separately to ease the burden on speculative versioning mechanisms. Dahlgren et al. [6] analyzed the efficacy of write caches in parallel architectures supporting relaxed consistency models and demonstrated major improvements in miss penalties associated with coherence misses. While the study is not directly related to TM, the results therein suggest that transactional semantics permit flexibility in handling updates issued within atomic code blocks. Dice et al. [7] mention the use of store buffers to confine transactional updates in the Rock processor which provides limited support for TM constructs. However, the effects of buffering speculative stores and its impact of HTM has been touched upon in few prior studies.

Several prior studies have looked at migratory sharing optimizations in scenarios other than HTM [5], [11], [23]. These studies inspired write-set prediction in LogTM [2], [15]. However, as shown in this study, TM performance is very sensitive to this optimization, as it can hide available concurrency. Thus, when attempting to exploit parallelism using an inherently optimisitic approach like TM, it must be used very selectively. The design pre-

sented in this paper is able to do so by only applying this optimization to actually conflicting lines and reverting to normal behavior once the migratory nature of a cache block disappears.

7 Conclusions

In this work we have highlighted the importance of effective management of speculative updates in eager hardware transactional memory. Simple optimizations that can be applied locally are shown to have a large impact on performance of eager designs on high-contention workloads which, in the past, have been a weak spot for eager designs. Another important insight that can be inferred from this study is that when considering eager HTM designs, these local optimizations might be as important as system level HTM policy optimizations that have been the focus of study in most prior work on the topic. In particular, we have shown that accurate prediction and management of migratory cache blocks and avoiding stalls due to conflicting stores using special buffering can allow eager designs to exceed the performance of an idealized lazy design. This insight should be coupled with the observations that lazy HTMs are harder to integrate into existing designs and demand higher power due to more aggressive speculation. Thus, in the opinion of the authors, eager HTMs represent a more promising alternative for integration than lazy ones.

Acknowledgments

This work has been supported by the Swedish Foundation for Strategic Research under grant RIT10-0033, as well as by the Spanish MINECO under grant TIN2012-38341-C04.

References

[1] Manuel E. Acacio, José González, José M. García, and José Duato. The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *Proc. of the 11th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 155–164, 2002.

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

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

[4] Intel Corporation. Transaction synchronization extensions (TSX). In *Intel Architecture Instruction Set Extensions Programming Reference*, pages 506–529. February 2012.

[5] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, pages 98–108, 1993.

[6] Fredrik Dahlgren and Per Stenstrom. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26:193–210, 1995.

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

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

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

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

[11] Stefanos Kaxiras and James R. Goodman. Improving cc-numa performance using instruction-based prediction. In *Proc. of the 5th Symp. on High-Performance Computer Architecture*, page 161, 1999.

[12] Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proc. of the 18th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 293–302, 2009.

[13] 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, 2002.

[14] 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, 2005.

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

[16] Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. Eager meets lazy: The impact of write-buffering on hardware transactional memory. In *Proc. of the 40th Int'l Conf. on Parallel Processing*, 2011.

[17] Anurag Negi, Rubén Titos-Gil, Manuel E. Acacio, Jose M. Garcia, and Per Stenstrom. π-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proc. of the 18th Symp. on High-Performance Computer Architecture*, 2012.

[18] Martin Ohmacht. Hardware support for transactional memory and thread-level speculation in the IBM Blue Gene/Q system (keynote speech). In *Workshop on Wild and Sane Ideas in Speculation and Transactions, co-located with PACT,*, 2011.

[19] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero, and Sourav Roy. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *Proc. of the 11th Int'l Conf. on High Perf. Computing and Comm.*, pages 171–179, 2009.

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

[21] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proc. of the 35th Int'l Symp. on Computer Architecture*, pages 139–150, 2008.

[22] Jonas Skeppstedt and Per Stenström. Simple compiler algorithms to reduce ownership overhead in cache coherence protocols. In *Proc. of the 6th Int'l Symp. on Architectural Support for Programming Language and Operating Systems*, pages 286–296, 1994.

[23] Per Stenström, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proc. of the 20th Int'l Symp. on Computer Architecture*, pages 109–118, 1993.

[24] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proc. of the 42nd Int'l Symp. on Microarchitecture*, pages 145–155, 2009.

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

Rubén Titos-Gil received his MS and PhD degrees in computer science from the University of Murcia, Spain, in 2006 and 2011, respectively. He is currently Postdoctoral Research Associate at Chalmers University of Technology, Sweden. His research interests lay on the fields of parallel computer architecture and programming models, including synchronization, coherence protocols and memory systems. He was a Spanish MEC-FPU Fellowship recipient from 2007 to 2011. He is a member of the IEEE.

Anurag Negi is a Ph.D. candidate at Chalmers University of Technology. His research interests include computer architecture, multicore memory hierarchies and support for transactional memory in hardware. In the past he has worked for Intel and Conexant Systems in India. He has bachelor's and master's degrees in electrical engineering from the Indian Institute of Technology, Madras.

Manuel E. Acacio is an Associate Professor of computer architecture and technology at the University of Murcia, Spain, where he leads the Computer Architecture & Parallel Systems (CAPS) research group. He has served as a committee member of important conferences, ICPP and IPDPS among others, and is currently an Associate Editor of IEEE Transactions on Parallel and Distributed Systems (TPDS). Dr. Acacio is actively working on prediction and speculation in multiprocessor memory systems, synchronization in CMPs, power-aware cache-coherence protocols for CMPs, fault tolerance, and hardware transactional memory systems. He is a member of the IEEE.

José M. García received a MS degree in Electrical Engineering and a PhD degree in Computer Engineering both from the Technical University of Valencia in 1987 and 1991 respectively. He is professor of Computer Architecture at the Department of Computer Engineering, and also Head of the Parallel Computer Architecture Research Group. His current research interests lie in high-performance power-efficiency coherence protocols for Chip Multiprocessors (CMPs), interconnection networks and general-purpose computing computing on GPUs. He has published more than 150 refereed papers in different journals and conferences in these fields. Prof. García is member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. He is also member of several international associations such as the IEEE and ACM.

Per Stenstrom is professor at Chalmers University of Technology. His research interests are in high-performance computer architecture. He has authored or co-authored three textbooks and more than a hundred publications. He has been program chairman of the IEEE/ACM Symposium on Computer Architecture, the IEEE High-Performance Computer Architecture Symposium, and the IEEE Parallel and Distributed Processing Symposium. He is an editor of ACM TACO and Associate Editor-in-Chief of JPDC. He is a Fellow of the ACM and the IEEE and a member of Academia Europaea and the Royal Swedish Academy of Engineering Sciences.