FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery *

Marc Lupon
Universitat Politècnica de Catalunya
mlupon@ac.upc.edu

Grigorios Magklis
Intel Barcelona Research Center
Intel Labs-UPC
grigorios.magklis@intel.com

Antonio González
Intel Barcelona Research Center
Intel Labs-UPC
antonio.gonzalez@intel.com

***Abstract*—Version management, one of the key design dimensions of Hardware Transactional Memory (HTM) systems, defines where and how transactional modifications are stored. Current HTM systems use either eager or lazy version management. Eager systems that keep new values in-place while they hold old values in a software log, suffer long delays when aborts are frequent because the pre-transactional state is recovered by software. Lazy systems that buffer new values in specialized hardware offer complex and inefficient solutions to handle hardware overflows, which are common in applications with coarse-grain transactions.**

In this paper, we present FASTM, an eager log-based HTM that takes advantage of the processor's cache hierarchy to provide fast abort recovery. FASTM uses a novel coherence protocol to buffer the transactional modifications in the first level cache and to keep the non-speculative values in the higher levels of the memory hierarchy. This mechanism allows fast abort recovery of transactions that do not overflow the first level cache resources.

Contrary to lazy HTM systems, committing transactions do not have to perform any actions in order to make their results visible to the rest of the system. FASTM keeps the pre-transactional state in a software-managed log as well, which permits the eviction of speculative values and enables transparent execution even in the case of cache overflow. This approach simplifies eviction policies without degrading performance, because it only falls back to a software abort recovery for transactions whose modified state has overflowed the cache.

Simulation results show that FASTM achieves a speed-up of 43% compared to LogTM-SE, improving the scalability of applications with coarse-grain transactions and obtaining similar performance to an ideal eager HTM with zero-cost abort recovery.

***Keywords*-harwdare transactional memory; transactional coherence protocols; fast abort recovery; FASTM**

I. INTRODUCTION

A high performance Transactional Memory (TM) system must provide an efficient implementation of the mechanisms that guarantee transactional semantics, offering fast execution in case of infrequent conflicts and minimizing the impact of collisions among transactions when contention is present.

Software Transactional Memory (STM) systems implement these mechanisms in software [1], Hardware Transactional Memory (HTM) systems opt for hardware implementations [2]–[9] while Hybrid Transactional Memory (HyTM) systems use a combination of the two [10], [11]. While STM systems provide flexibility and portability, HTM and HyTM systems provide accelerated implementations with significantly lower overheads.

Version management is one of the key design dimensions of a TM system, together with conflict detection and conflict resolution. Version management defines how and where transactional modifications are stored and what actions must be performed at commit and abort time. Current HTM systems fall into one of two distinct strategies for version management: eager or lazy [12].

Lazy version management [4], [6], [13] keeps old (pre-transactional) state in-place in memory and buffers new state (values generated inside the running transaction) elsewhere. This makes aborts fast, but commits have an overhead because the new state must become globally visible. Most lazy systems use the L1 caches to buffer new state, and specialized coherency protocols [4], [7] to hide transactional updates from the rest of the memory hierarchy. Other implementations, like the one proposed for Rock [13], store transactional modifications in a gated store buffer, the content of which is drained at commit time.

In case the new state overflows its buffering space, some lazy HTM systems behave similar to HyTM systems, and fall-back to a STM implementation. Some other lazy HTMs, such as LTM [3], VTM [6] or FlexTM [7], store overflowed state in a data structure kept in memory, which must be accessed on cache misses and on commits. Falling-back to STM incurs significant performance loss while fully-hardware HTMs require complex and cumbersome hardware mechanisms. This makes transactional state overflows the main drawback of lazy version management systems.

On the other hand, eager version management [5] puts new state in-place in memory and buffers pre-transactional state elsewhere, usually a software-managed log structure in

* This work is supported by the Spanish Ministry of Science and Innovation and FEDER funds of the EU under contracts TIN 2007-61763 and Intel Corporation. Marc Lupon is supported by an UPC-Research grant

	Hardware Support	Abort Recovery	Overflow Policy	Commit Process
LogTM-SE [14]	Logging	Software	Update Memory	-
Rock HTM [13]	Store Buffer	Hardware	Notify Software	Drain Buffer
HyTMs [10], [11]	L1 TX Cache	Hardware	Run STM	Update Memory
FlexTM [7]	L1 TX Cache	Hardware	Software Structure	Update Memory
FASTM	L1 TX Cache	Hardware Software	Update Memory	Clean L1 State

Table I
CHARACTERISTICS OF HTM SYSTEMS

cacheable memory [5], [8], [9], [14]. This makes commits fast, since data is already stored in memory, but aborts have an overhead because the old state must be recovered.

Also, since the pre-transactional state is stored in the log and can be recovered, transactional modifications can be put anywhere in the memory hierarchy, so eager systems do not suffer from cache/buffer overflows like lazy ones. LogTM-SE [14] is an example of an eager HTM. Table I summarizes the main characteristics of several state-of-the-art HTM systems.

Previous studies [15] have claimed that common-case transactions were short and did not usually conflict. However, newer, more complex workloads that are believed to better represent future transactional applications [16] exhibit a significant number of large and/or conflicting transactions. The execution of large transactions has uncovered performance issues with current implementations of both eager and lazy version management HTMs.

Eager log-based systems suffer considerable delays in the execution of conflicting large transactions due to the overheads of abort recovery [17], [18]. Moreover, slow aborts may exacerbate contention, as many conflicts involve transactions in their abort-recovery phase, which in turn provokes more aborts. In lazy HTMs, the overflow mechanism becomes critical given that cache evictions are common in coarse-grain transactions that access a large number of cache lines, as we will show in Section IV, where we present our experiment results.

All of the above has led us to develop FASTM, a log-based HTM with eager version management that keeps both the new state and the pre-transactional state in memory to provide fast commits and aborts. FASTM achieves this by pinning down new values in the L1 caches, similar to lazy version management systems, but with two key differences: (a) transactions update memory in-place, so commit requires no special actions, and (b) overflows are handled gracefully by using a software-managed log, like eager version management systems.

In FASTM, we also change the cache coherence protocol and the L1 cache controller, to guarantee that if there are no overflows, the old state is in-place in the higher levels of the memory hierarchy. Aborts in FASTM are fast, because they only require the invalidation of the L1 transactional lines. On the other hand, since the pre-transactional values are kept in a log on the side, if a transactionally modified line is evicted from the cache the system can recover the old values from the log (using the software abort mechanism).

Our evaluation of FASTM shows that our proposal achieves a speed-up of 43% on average compared to LogTM-SE, a state-of-the-art eager log-based HTM. Our analysis shows that FASTM substantially accelerates applications with coarse-grain transactions, because it minimizes the time spent on abort recovery and it reduces the number of conflicts without losing performance in case of cache overflows. In fact, our approach achieves similar performance to an idealized eager HTM system with instantaneous (zero-latency) abort recovery.

The remainder of the paper is organized as follows. In Section II, we give an overview of the FASTM system, whereas in Section III we describe in detail the basic transactional operations of FASTM. In Section IV, we show the results of the evaluation of our proposal. In Section V, we present related work in HTM. Finally, in Section VI, we conclude the paper.

II. THE FASTM SYSTEM

FASTM is an eager HTM system based on LogTM-SE [14], so our proposal requires mostly the same hardware support. FASTM uses two hardware signatures to track transactional accesses: a *Read* signature to identify read conflicts and a *Write* signature to detect write conflicts in case of overflow. Also, it keeps a software log in the same way as log-based systems: each transactional store copies the old value to the log before updating the memory with the new value. We assume that logging is a dual-phase process where, (1) the old line is brought to the processor and is written in the first free entry of the log and (2) the new value is stored in the cache. The combination of the signatures and the software log allows FASTM to gracefully handle cache overflows.

The novelty in FASTM is in the way it manages the transactional state and in its abort recovery mechanism. Following the example of many lazy version management systems [6], [7], FASTM utilizes a new coherence protocol for the L1 cache (we call it TMESI).

TMESI is a write-back protocol that provides fast commits because it does not hide transactional updates from the memory hierarchy—FASTM is an eager version management system—but it does enforce the following condition: transactionally modified lines are "pinned" in the L1 cache (they cannot write back) to guarantee that a valid copy of the pre-transactional version of the line exists in the memory hierarchy until commit/abort time (or until an overflow occurs). This operation is similar to some Thread-Level Speculation protocols [19].

	Trans (T)	Dirty(D)	Valid(V)
T	1	1	0
M	0	1	0
E	0	1	1
S	0	0	1
I	0	0	0

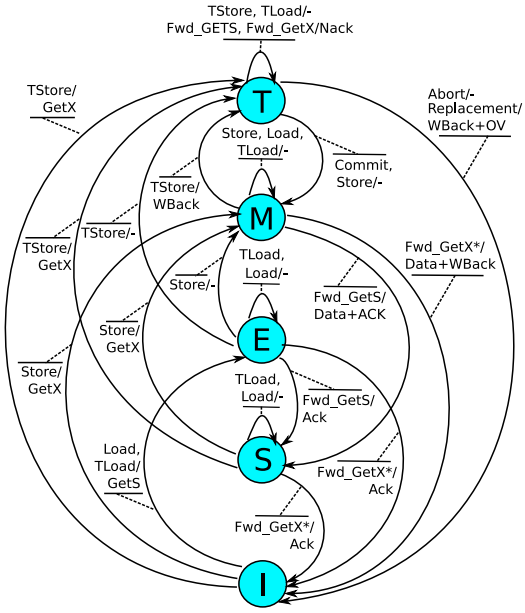

Figure 1. TMESI cache transition diagram

To allow this special handling of transactional stores, TMESI modifies the classical MESI protocol to put said lines to a new state, named *T*, where they persist until the transaction commits, aborts or overflows. Therefore, FASTM requires, as it is shown in Figure 1, an extra bit to encode the *T* state and some logic to identify transactional stores.

These *T* lines are also used to detect conflicts among transactions, so the *Write* signature only contains the addresses of lines that overflow (get evicted from) the L1 cache. This fact reduces the aliasing in the *Write* signature, increasing its fidelity.

With the TMESI protocol, the system guarantees that if no overflow occurs, the old values are still in-place in the higher levels of the memory hierarchy. If a transaction that has not overflowed the L1 cache aborts, FASTM provides a very fast abort mechanism: it simply invalidates the lines modified by the transaction (this is a silent invalidation, more on this later).

If an overflowed transaction aborts, FASTM falls back to a software recovery mechanism similar to that employed in LogTM-SE. The software abort recovery process requires just a few registers to hold the last entry of the log, the address of the abort recovery routine and the Program Counter (PC) of the current transaction.

Like most of the eager HTM implementations, FASTM performs eager conflict detection and eager conflict resolution. FASTM borrows the conflict detection engine from LogTM-SE, where the directory forwards transactional requests to the private caches. Moreover, as it is described in Section IV-E, FASTM supports multiple resolution policies.

Figure 1 shows the principal state transitions of the TMESI coherence protocol. In the diagram, the triggering message is written before the slash and its associated action after ('–' means none). *TStore* and *TLoad* are memory accesses produced inside a transaction. *Fwd_GetS* is a directory forwarding load request from a remote processor, *Fwd_GetX** is a forwarding write request without a conflict. In case of conflict (*Fwd_GetX*), the line remains in the same state, sending a *Nack* to the requester. *WBack* action pushes the line to the higher levels of the memory hierarchy. *Replacement* indicates a cache eviction. In the case of replacing a transactional line, a set of overflowed actions are required (*OV* actions). Detailed explanation of the TMESI transitions is presented in Section III.

III. FASTM TRANSACTIONAL OPERATIONS

This section describes how FASTM operates, explaining in detail how transactional lines interact with the system. We present the basic operations of the system and describe cache replacements and the mechanism for abort recovery. For our discussion we will assume a CMP system with single-threaded cores and two levels of caches, where the L1 is private per core and the L2 is shared. Coherency is implemented using a directory at the L2 cache.

A. Transactional Loads

Assume a core *C*0 that performs a transactional read (*TLoad*) operation. In FASTM, *TLoads* are performed as regular loads. However, in order to maintain transactional coherency, the *TLoad* address must be added to the *Read* signature of *C*0, which is used to detect conflicts with remote transactional stores. *C*0 only has to check for conflicts when loading a line that is not present in its L1 cache. In this case, *C*0 must request the line from the directory in the L2 cache, which serves the line if there are no writers. If there is a writer, the directory forwards the request to the core that owns the line (assume core *C*1).

When *C*1 receives the forwarding read request (*Fwd_GetS*), *C*1 must acknowledge it. If *C*1 has the line in its L1 cache in *T* state (*i.e.*, it is a transactional, non-oveflowed line) then it sends a *Nack* reply to *C*0 and the conflict is resolved according to the conflict resolution policy.

If the requested line is not in *T* state or it is not in *C*1's L1 cache, then *C*1 must check its *Write* signature. This is necessary to guarantee coherence for transactions that overflow the cache. If there is a match in the signature, *C*1 replies to *C*0 with a *Nack*. Otherwise, the line moves to the *S* state and, if the line was previously in the *M* state, *C*1 forwards the data to *C*0 and also writes it back to the L2 (this is the same as in a typical MESI).

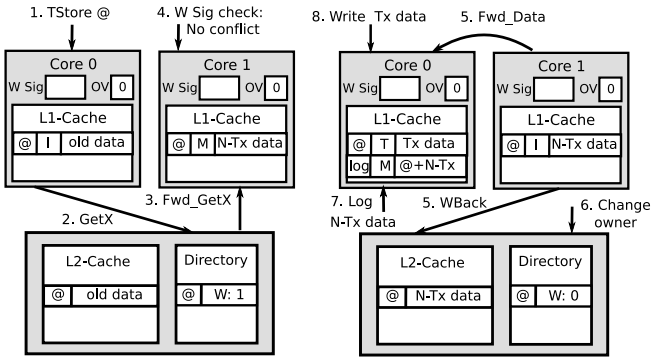

Figure 2. Transactional store operation

B. Transactional Stores

Assume a core $C0$ that performs a transactional store (*TStore*) operation. If $C0$ has the line in its L1 cache in an exclusive (T or E) state, it changes the cache state to T and the *TStore* completes immediately.

If the line was previously written by $C0$ inside a transaction that has already committed, or by non-transactional code, the line may be in $C0$'s cache in the M state. If so, then $C0$ must write-back the line data to the L2 before transitioning the line to the T state and completing the *TStore*. This write-back does not generate any coherence requests to the other L1 caches, but it is necessary to guarantee that the L2 always has the correct pre-transactional state.

In Figure 2 we can see and example of the case where $C0$ misses in the cache (having the line in the S state is identical). The left (right) of the figure shows the state of the system before (after) the *TStore*. When $C0$ misses in its L1 (step 1), it requests the line from the directory (step 2), and the directory forwards the request (*Fwd_GetX*) to the line current owner, in this case $C1$ (step 3).

If $C1$ has the line in the T state, it *Nacks* the request directly without checking the signatures. Otherwise, it checks its *Read* and *Write* signatures to detect conflicts with the requesting transaction (step 4). If a positive match is found, $C1$ *Nacks* the request from $C0$ and the conflict resolution mechanism kicks in. If the line is not being accessed by any transaction (*i.e.*, $C1$ has it in M, or E state), the directory gives the ownership to $C0$, invalidating all other copies of the line (in this case $C1$).

With MESI, if $C1$ has the line in M state it must forward the line data to $C0$ before $C0$ can become the new owner of the line. In TMESI, the L2 cache must always have a copy of the old value in order to guarantee correct abort recovery for non-overflowing transactions. For this reason, $C1$ also sends a copy of the forwarded line to the L2 (step 5) before relinquishing ownership of the line to $C0$ (step 6), allowing $C0$ to safely write the transactional value (step 7-8).

C. Transactional Cache Replacements

Assume a core $C0$ that replaces a line with transactional modifications. In FASTM, cache evictions of lines in T state write back the speculative values to the higher levels of the memory hierarchy, similar to evictions of lines in M state. This is analogous to other eager log-based HTM systems, and it is safe to do because the pre-transactional values are kept in a software log. Nonetheless, the system must perform some actions before pushing the speculative data to the L2 cache.

First, the evicted line address must be added to $C0$'s *Write* signature. The directory maintains as the owner of the line the current core ($C0$) and will forward all future remote requests to it. As discussed earlier, upon receiving a remote request $C0$ will check its *Read* and/or *Write* signatures to discover conflicts. If the $C0$ evicted line is also replaced from the L2 cache, the request is forwarded to all the processors, which must check their signatures. This fact permits the conflict detection engine to identify collisions that involve evicted transactionally written lines.

Second, a transaction overflow flag in $C0$ is asserted to inform the processor that the transaction has to be aborted by software. In FASTM, we have chosen to write all the updated lines in the software log, to allow software abort recovery.

An alternative, is to only insert overflowed lines in the software log (instead of all updated lines). This approach is more efficient, because it reduces abort recovery time of overflowed transactions, given that fewer lines must be restored by the software routine. Moreover, this results to less cache pollution (the software log is in cacheable memory) which may result in less transactional evictions.

However, this hybrid solution complicates the abort recovery mechanism, which must maintain the atomicity of a dual phase hardware/software abort. The upside of maintaining the software log for all updated lines is that it allows the use of mechanisms like those of LogTM-VSE [20] to survive context switches or page faults.

D. Committing Transactions

FASTM provides, like other eager HTM systems, a fast commit, even for overflowed transactions. FASTM only commits consistent transactions, therefore no additional actions are needed to guarantee consistency. In FASTM, a committing transaction first flush-clears the T bit of all cache lines, moving all T lines to M, and then releases the signatures. Notice that replaced lines do not require any commit action, because transactional modifications are already in the memory hierarchy. In contrast to lazy version management schemes [4], our system does not require sending state updates to the directory. Instead, the directory already has the committer as the owner of the line (it acquired ownership during the execution of the transaction).

E. Aborting Transactions

FASTM uses a hardware-accelerated abort recovery mechanism for non-overflowed transactions and a software abort recovery mechanism for transactions that have evicted lines in the *T* state. The processor decides which of the two recovery mechanisms applies by checking its overflow flag.

Non-overflowed transactions use the coherence protocol to discard transactional modifications. This process is performed by silently invalidating all the *T* state lines in the L1 (the directory is updated lazily by future requests). Hence, when the transaction restarts again, it must re-acquire the ownership of each line. This can be safely done because the L2 cache keeps the pre-transactional state.

Assume core $C0$ aborts and now core $C1$ requests a line that $C0$ wrote inside the aborted transaction. First, the directory will forward the request to $C0$ since it is still the owner. $C0$ acknowledges the request, informing $C1$ that it ($C0$) is no longer the owner. Then, $C1$ will take the line from the L2 instead, which still keeps the pre-transactional value, and the directory will be updated. This lazy directory update removes the communication with shared resources, allowing a fast abort recovery.

The invalidation of *T* state lines increases the number of L1 misses on restarted transactions. However, this situation is not critical mainly for two reasons. First, most transactions have considerably smaller write sets than read sets, so the rate of L1 misses is not a bottleneck (read lines are not invalidated in FASTM). Second, these L1 misses are served faster than conventional L1 misses because these lines are still owned by the aborted transaction.

Assume core $C0$ aborts and tries to re-acquire a line invalidated by the fast abort mechanism. $C0$ requests the line from the directory, which still has $C0$ as its owner. Thus, the line can be directly served from the L2 cache, without requiring coherency operations or signature checking.

Transactions that overflow the L1 are recovered by software by taking a trap to the recovery handler. The recovery handler is a software routine that walks the log in reverse order and, for each entry, writes the logged data to its correspoding place in memory. Notice that some of the *T* state lines may be overwritten by the recovery handler. Such writes are performed by non-transactional stores, moving the lines from *T* to *M*. When the software abort-recovery mechanism finishes, it returns control to the hardware.

Both the hardware and the software mechanisms release the signatures when the recovery process finishes.

IV. EVALUATION

For the evaluation of FASTM we assume a Chip Multiprocessor (CMP) with 16 cores, as shown in Figure 3. The system has a 16-node mesh interconnect that uses 64-byte links with adaptive routing, where each node has a core, a 1 MB shared L2 cache and part of the directory. This is a Non-Uniform Cache Access (NUCA) system, where the L2 cache is distributed among the cores. The system has four memory controllers to access main memory. Each core has two 2 Kbit signatures (*Read* and *Write*) to track transactional memory accesses. Detailed system parameters are shown in Table II.

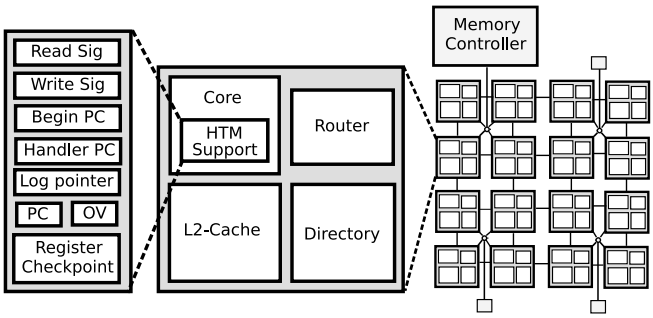

Figure 3. System scheme with HTM support

Core	1.2 GHz in-order, single issue, single-threaded
L1 cache	32 KB 4-way, 64-byte line, write-back, 2-cycle latency
L2 cache	16 MB 8-way, banked NUCA, write-back, 15-cycle latency
Memory	4 GB, 4 banks, 150-cycle latency
L2 directory	Bit vector of sharers, 6-cycle latency
Interconnect	16-node Mesh, 64-byte links, 2-cycle wire latency, 1-cycle router latency
Signatures	2 Kb Parallel Chuckoo-Bloom filters

Table II
BASE SYSTEM PARAMETERS

The base system and the coherence protocols have been simulated using the Simics [21] simulation infrastructure from Virtutech and the GEMS [22] toolset from Wisconsin's Multifacet group. For our analysis we use applications from the SPLASH-2 [23] and the STAMP benchmark suites [16], and two microbenchmarks from the GEMS 2.0 distribution.

Table III provides important information about the applications we utilize. The first three columns show the benchmark suite, the application name, and its input parameters. The fourth column (Tx Time) shows the time spent inside transactions as a percentage of the total execution time, and the next column (Cycles Tx) shows the average number of cycles per transactions. These numbers were collected in single-thread FASTM execution.

Table III also shows the classification of the applications by with the granularity of their transactions. Fine-grain applications (top half of Table III) spent most of their time in non-transactional code and in small transactions, which usually scale well. On the other hand, coarse-grain applications (bottom half of Table III) spent most of their time in big transactions that suffer important performance penalties when they conflict.

A. HTM Base Systems

For our analysis we have chosen to compare FASTM with two other eager version management HTM systems,

						LogTM-SE		FASTM			
Suite	Bench	Input parameters	Tx Time	Cycles Tx	Commit	Abort Rate	Abort Confl	Abort Rate	Abort Confl	Tx OV	SW Abort
μbench	Btree	50% insertions, 100K Tx	49.75%	731.08	100000	0.71	56.7%	0.65	0%	$\approx$ 0%	0%
	Deque	5K dummy work, 100K Tx	1.13%	88.72	100000	2.12	53.1%	0.24	0%	0%	0%
Splash-2	Barnes	512 bodies	2.23%	617.18	2362	0.61	62.2%	0.49	0%	0.4%	0%
	Raytrace	teapot	0.15%	8.33	47766	0.73	49.6%	0.15	0%	0%	0%
STAMP	Kmeans	15/15 clusters, 16K points	9.14%	1513	21846	0.06	62.1%	0.02	0%	0.1%	0%
	Ssca2	2^{14} nodes, 9 edges, 9 length	9.27%	179.26	93684	0.1	$\approx$ 0%	$\approx$ 0	0%	0%	0%
	Bayes	32 vars, 1024 records	81.63%	63852	490	4.01	29.2%	1.92	0.4%	12.9%	5.7%
	Genome	64K seg, 1K gene, 32 length	97.46%	5369	40037	0.16	40.3%	0.13	$\approx$ 0%	0.24%	$\approx$ 0%
	Intruder	4K traffic, 10 attacks, 4 pack	42.8%	1403	22500	3.34	52.5%	2.59	0.51%	0.85%	$\approx$ 0%
	Labyrinth	32*32*3 maze, 1024 routes	99.76%	97910	2048	2.31	68.3%	0.26	0%	17.3%	5.3%
	Vacation	64K entries, 4K tasks, high	89,62%	18775	4096	0.18	12.6%	0.1	0	3.24%	0.26%
	Yada	20 angle, 633.2 input mesh	99.92%	14203	2788	2.18	6.97%	2.06	$\approx$ 0%	12.3%	0.29%

Table III
FINE-GRAIN (TOP) AND COARSE-GRAIN (BOTTOM) BENCHMARK CHARACTERIZATION

although with different underlying mechanisms. The first one, which serves as our baseline, is LogTM-SE, particularly the implementation that is distributed with GEMS 2.0 [22]. The second one, is an idealized eager version management HTM that servers as our upper-bound.

As explained earlier, LogTM-SE [14] keeps older values with their respective addresses in a software log, which is traversed by software in case of abort. The idealized system is similar to an eager HTM, but it provides zero-latency abort recovery. We emulate this behavior by modifying LogTM-SE to use an infinite hardware buffer to keep the log, and by allowing the entire buffer to drain in a single cycle. Moreover, the idealized implementation uses perfect signatures.

Both LogTM-SE and the ideal implementation use a MESI coherence protocol with signature checking to detect conflicts among transactions. In contrast, FASTM uses TMESI to restore the pre-transactional state.

We have used the **Stall** conflict resolution policy for the comparisons between LogTM-SE and FASTM. Stall is the policy implemented by LogTM-SE [14]. After detecting a conflict between two transactions, this policy stalls the requester, who waits until the other transaction commits. However, to avoid cyclical dependences among stalled transactions, transactions must inform a centralized cycle-detector when they are stalled. If a dependence cycle occurs, a timestamp determines the younger transaction that participates in the cycle and aborts it. After recovery, an exponential backoff is performed to guarantee progress.

We decided to use the Stall conflict resolution policy for all the comparisons between LogTM-SE and FASTM for two main reasons. First, this policy minimizes the number of aborts, which become critical in an HTM with software abort recovery (also, by minimizng aborts we are conservative in how much FASTM improves over LogTM-SE). Second, by using the Stall policy for our evaluation it is easier to compare our results with previous LogTM-SE characterizations [12], [14], [17], [18]. In Section IV-E we describe other conflict resolution policies and we discuss about how they behave in LogTM-SE/FASTM.

Moreover, we have also evaluated FASTM-Sig, a variation of FASTM where all *TStore* addresses are added to the *Write* signature (remember that FASTM only updates the *Write* signature with the *T* state lines that get evicted). Studying this alternative allows us to determine the performance benefits of reducing aliasing in the signatures.

B. Performance Analysis

Figure 4 presents the time distribution of LogTM-SE (labeled L), FASTM (labeled F) and Ideal (labeled I) HTM systems in their 16-threaded executions using the Stall conflict resolution policy. The execution time has been normalized to the 16-threaded LogTM-SE execution and is broken down to: non-transactional and barrier cycles (labeled Non-Tx and Barrier), the time spent in committed transactions (labeled Good Tx), the time that is wasted in non-useful work discarded from aborted transactions (labeled Aborted Tx), the time spent in abort recovery (labeled Aborting), the time that transactions remain stalled waiting for a conflict to be resolved (labeled Stalled), and the time that processors execute the exponential backoff after aborting (labeled Backoff).

As it can be seen in Figure 4, FASTM has an average speed-up of 43% over LogTM-SE, achieving similar performance to the ideal approach. The benefit is especially notable in some coarse-grain applications, where FASTM obtains more than 5X speed-up with respect to LogTM-SE. The reasons why FASTM outperforms LogTM-SE in all the benchmarks are explained in the following paragraphs.

First, FASTM decreases the time spent in abort recovery, which reduces overall execution time. As we can see in Figure 4, the LogTM-SE recovery mechanism accounts for 5.6% of the total execution time on average. However, in coarse-grain applications, like *Intruder* or *Labyrinth*, up to 15% of the time is spent in the software abort routine. This undesirable overhead can be reduced if we apply a fast

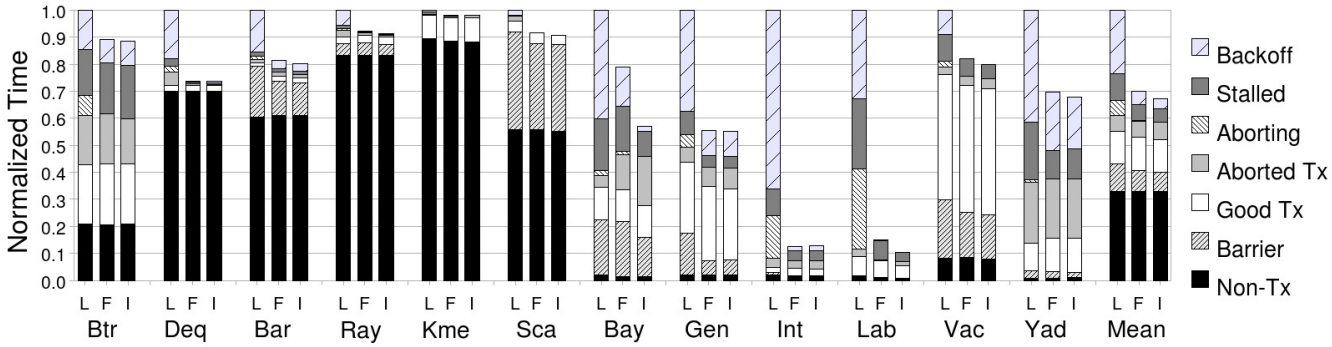

Figure 4. 16-threaded normalized distribution time of HTM systems

abort recovery mechanism. In fact, FASTM only spends, on average, 0.2% of the execution time to restore the pre-transactional state.

Second, by reducing the abort recovery time, FASTM decreases the number of conflicts that involve transactions in their abort recovery phase. In LogTM-SE, the transaction is alive until the very end of the abort recovery procedure. Thus, remote transactions that want to access to data owned by the aborting transaction will generate conflicts. As FASTM aborts transactions faster, most of the conflicts produced in the LogTM-SE abort period disappear. This benefit can be seen from the data in Table III, which shows, for both HTM systems, the rate of aborts per transaction (labeled Abort Rate) and the percentage of aborts caused by, at least, one transaction that is aborting (labeled Abort Confl).

Although LogTM-SE does not lose much performance in fine-grain applications due to their parallel nature, some high-contention benchmarks, like *Deque* or *Barnes*, are far from the Ideal because more than 15% of the execution time is devoted to conflict management. Coarse-grain applications that have lots of aborts, like *Bayes*, *Intruder* or *Yada*, also require a large number of backoff or stall cycles (up to 60%) in LogTM-SE. In these benchmarks, the fast abort recovery of FASTM reduces the time wasted in non-useful transactional work, the time spent in stalled transactions and the time that processors execute the backoff.

C. Scalability Analysis

Figure 5 shows the scalability of 16-threaded applications run with LogTM-SE, FASTM, FASTM-Sig and Ideal. The baseline is a single-threaded LogTM-SE execution. As it can be seen, fine-grain applications that execute small transactions exhibit good scalability in the majority of TM systems given that most of their time is spent in non-transactional code, except for *Raytrace* and *Ssca2*, where scalability is lower because threads must wait in barriers.

Genome and *Vacation* are coarse-grain applications that scale well because they present few aborts. However, other applications with large transactions do not scale because most of the transactions conflict or overflow. This puts a lot of pressure on the version management mechanism and the conflict resolution policy. Benchmarks like *Genome*, *Intruder* or *Labyrinth* scale poorly with LogTM-SE because large transactions are recovered by software. In contrast, FASTM significantly improves their scalability, obtaining significant speed-up with respect to the single-threaded execution.

FASTM can also take advantage of the cache *T* state to detect conflicts and to reduce the pressure on signatures, which may lead to less false conflicts. However, this fact is not critical in the majority of the benchmarks. As can be seen in Figure 5, benchmarks with small or medium size transactions do not suffer from false positives when 2 Kbit signatures are used. Only *Labyrinth*, which executes huge transactions, gains from this enhancement, showing a speed-up of 16% in FASTM vs FASTM-Sig.

On the other hand, FASTM-Sig facilitates the use of mechanisms like those of LogTM-VSE [20] to survive context switches or page faults (because the write set of the transaction is already in the *Write* signature). With FASTM, the *Write* signature has to be reconstructed from the log (the hardware *Write* signature does not include the *T* state lines in the L1). Given that our evaluation shows that the fidelity of the *Write* signature is not critical, FASTM-Sig may be a good alternative to simplify transaction virtualization.

D. Overflow Analysis

Figure 5 shows that, in fine-grain applications, FASTM achieves similar performance to the ideal eager implementation. This is because fine-grain applications almost never evict transactional cache lines, so no software aborts are performed. This can be seen from the data in Table III, where we can see the number of committed transactions (labeled Commit) and the percentage of transactions that evict transactional lines from the L1 cache (labeled Tx OV). We can also see the percentage of aborts that are restored by software (labeled SW Abort).

Although some coarse-grain benchmarks, like *Bayes*, *Labyrinth*, *Vacation* or *Yada*, have an important number

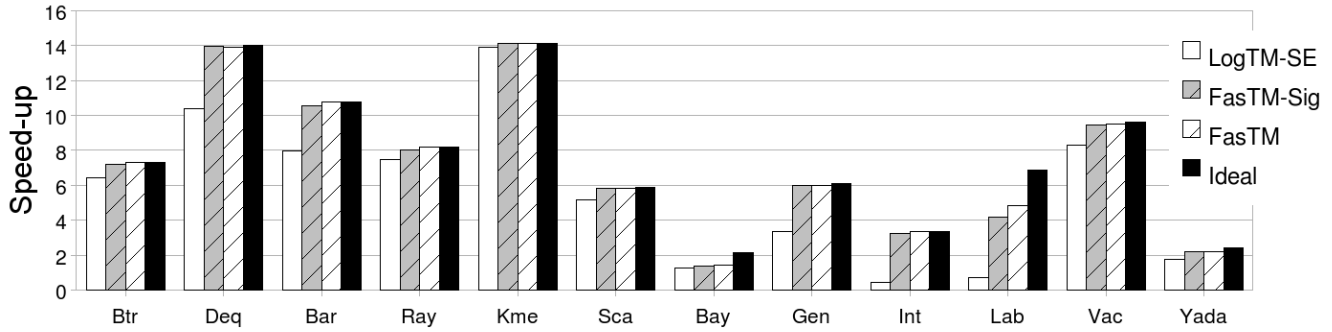

Figure 5. 16-threaded scalability of HTM systems

of overflows, FASTM recovers the majority of the aborted transactions almost immediately by hardware. Therefore, FASTM performs similar to the ideal implementation for most of the benchmarks. The only exceptions are *Bayes* and *Labyrinth*, which still suffer a significant amount of software aborts and false conflicts caused by finite signatures.

E. Conflict Resolution Analysis

The Stall conflict resolution policy sometimes exhibits pathological behavior that can affect the performance of the application [12]. For this reason, we have evaluated both LogTM-SE and FASTM with three other conflict resolution policies:

Abort: Aggressive policy that tries to eliminate the conflicts generated by stalled transactions. When a conflict is detected, the system aborts the requester, instead of stalling the transaction [24]. It also requires a backoff to avoid multiple aborts of transactions.

Timestamp: Policy that eliminates the backoff cycles by guaranteeing the progress of the oldest transaction, based on [24]. If a processor receives a conflicting request, it checks the remote timestamp and, if it is older than the local timestamp, the processor aborts the local transaction after sending a *Nack* to the requester together with its timestamp. When a processor receives a *Nack*, it checks the remote timestamp and, if it is older than the local, it aborts the local transaction. Otherwise, it keeps issuing the request until the conflicting transaction finishes its abort recovery process.

Hybrid: Enhaced policy described as EE_{HP} in [12]. It works like the Stall policy, but write requests abort younger readers in order to eliminate starvation of the writer.

We have evaluated LogTM-SE with all the conflict resolution policies (the results are not shown due to space constraints), and we have found that the Stall policy outperforms the Abort and the Timestamp policy in LogTM-SE because it reduces the number of software aborts.

LogTM-SE with the Hybrid policy achieves better results than LogTM-SE with the Stall policy in benchmarks with small transactions and high-contention, like *Barnes* or *Deque*, or in applications with read-only transactions, like *Btree* or *Genome*. In these situations, LogTM-SE with Hybrid obtains similar performance to FASTM given that most aborted transactions do not need to restore too many lines. However, LogTM-SE with the Stall policy presents better performance in applications with large transactions, like *Vacation* or *Yada*.

FASTM can take advantage of aggressive conflict resolution policies because it minimizes the impact of aborts. Figure 6 shows the time distribution of FASTM with Stall (labeled S), Abort (labeled A), Timestamp (labeled T) and Hybrid (labeled H) conflict resolution policies normalized to the 16-threaded execution of FASTM with Stall.

The Abort policy removes stalling transactions in case of conflict given that transactions automatically abort. *Labyrinth* can benefit from this policy, because conflicts that involve stalled transactions disappear. However, in benchmarks with high-contention and small transactions, like *Barnes* or *Genome*, the number of aborts augments significantly, increasing the time spent in backoff.

The Timestamp policy improves some high-contention benchmarks with variable-size transactions, like *Genome* or *Vacation*, because it does not require backoff cycles. Nonetheless, the Timestamp policy has some weaknesses. First, it constantly aborts transactions, which increases considerably the discarded work in coarse-grain applications like *Bayes*. Second, a transaction remains stalled until the younger conflicting transaction finishes its abort phase. Although FASTM provides fast abort recovery, some transactions do not abort instantaneously. This problem is critical in benchmarks with software aborts, like *Labyrinth*.

The Hybrid policy improves our baseline because it reduces the starvation of older writers without increasing contention. Like in LogTM-SE, the Hybrid policy accelerates applications with high-contention and small/read-only transactions. Moreover, the fast abort recovery mechanism allows FASTM to improve the performance of some coarse-grain benchmarks as well, like *Intruder* or *Vacation*, which discard a lot of work when transactions abort.

V. RELATED WORK

Herlihy and Moss [2] introduced TM as a new programming paradigm that intended to make lock-free mechanisms

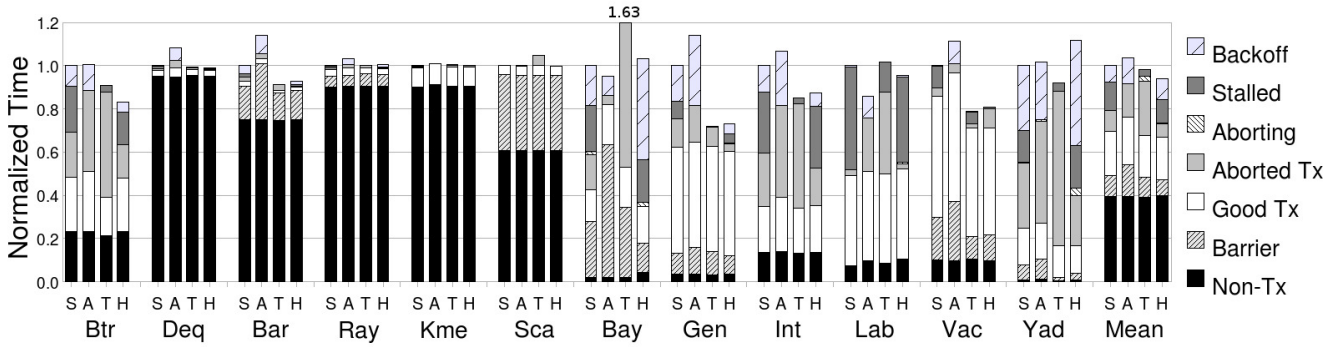

Figure 6. 16-threaded distributed time of FASTM conflict resolution policies

more efficient than blocking synchronization techniques. To this end, they included hardware transactional support in the microarchitecture, building a Hardware Transactional Memory (HTM) system.

Transactional Coherence and Consistency (TCC) [4] presented a new consistency model based on transactions, using the memory hierarchy to perform lazy data version management. A private cache buffers new values locally, while a second level cache, which is shared among processors, holds the old values. At commit time, transactions send all their modifications to the shared cache, making the changes visible to all the processors and propagating the write set to the rest of the processors, which abort their transactions in case of conflict.

Different proposals have been able to execute unbounded transactions using finite hardware. Hybrid Transactional Memories (HyTM) [10], [11] handle large transactions using software mechanisms [1], whereas common-case, smaller transactions use best-effort hardware.

Rock [13] will possibly be the first processor to include transactional hardware support. Rock stores transactional modifications in a gated store buffer, the content of which is drained at commit time. In case of buffer overflow, the system notifies an interruption and the software decides how the overflowed transaction is re-executed.

RTM [25] is a HyTM that modifies the cache coherence protocol to hide transactional updates in the L1. FlexTM [7] adapts the RTM protocol by adding two states to the typical MESI protocol. This fact allows the system to track the lines used in transactions and to implement a more flexible conflict management policy. Evicted transactional lines are buffered in a hash structure, called Overflow Table, which must be accessed by software to perform look-ups on cache misses and to ensure permanent commits. A similar overflow policy is implemented in LTM [3] or VTM [6].

UTM [3] proposed eager version management support for unbounded transactions, storing overflowed old lines in a software structure. This structure was walked to detect conflicts, to store old values or to undo transactional updates. LogTM [5] simplified this mechanism by storing old values and their associated address in a private log. Read-Write cache bits were used to detect conflicts and a software routine restored the pre-transactional state in case of abort.

Other eager log-based HTM proposals use the version management engine of LogTM. LogTM-SE [14] decouples transactional state from caches, replacing the Read-Write bits of LogTM with signatures [26]. OneTM [8] introduces a permission-only cache to maintain consistency of evicted cache lines. TokenTM [9] eliminates the false positives of signatures by adapting the concept of token coherence to detect conflicts among transactions.

VI. CONCLUSIONS

FASTM is the first eager version management HTM that, like lazy version management approaches, takes advantage of the processor's cache hierarchy to provide fast abort recovery. FASTM uses a novel coherence protocol to buffer the transactional modifications in the first level cache and to keep the non-speculative values in the higher levels of the memory hierarchy. This mechanism accelerates the abort recovery of large transactions, which is critical in eager log-based implementations like LogTM-SE.

To handle cache overflows, FASTM follows a log-based approach. Transactional cache lines are evicted in-place in the memory hierarchy and old values are maintained in a cacheable log, which must be restored by a software routine. This approach simplifies overflow mechanisms of lazy version management systems, that either need complex specialized hardware to handle cache misses and to commit overflowed lines or fall-back to software-only transactions.

We have evaluated FASTM with a heterogeneous set of applications and conflict resolution policies. Our proposal obtains, on average, a speed-up of 43% over LogTM-SE. We have seen that the performance improvement is more pronounced in applications with coarse-grain transactions, because FASTM reduces considerably the time spent in abort recovery as well as the number of conflicts. Although our analysis shows that transactional cache replacements are common in coarse-grain applications, FASTM does

not suffer performance penalties, because transactions that overflow the caches do not usually abort.

Our evaluation of FASTM with different conflict resolution policies shows that having a fast abort recovery mechanism favors aggressive policies that abort critical transactions in situations with high-contention.

REFERENCES

[1] N. Shavit and D. Touitou, "Software transactional memory," in *Procs. of the 14th ACM Symp on Principles of Distributed Computing*, Aug. 1995.

[2] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Procs. of the 20th Intl Symp on Computer Architecture*, May 1993.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Procs. of the 11th Intl Symp on High-Performance Computer Architecture*, Feb. 2005.

[4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Procs. of the 31st Intl Symp on Computer Architecture*, Jun. 2004.

[5] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Procs. of the 12th Intl Symp on High-Performance Computer Architecture*, Feb. 2006.

[6] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *Procs. of the 32nd Intl Symp on Computer Architecture*, Jun. 2005.

[7] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in *Procs. of the 35th Intl Symp on Computer Architecture*, Jun. 2008.

[8] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making The Fast Case Common And The Uncommon Case Simple In Unbounded Transactional Memory," in *Procs. of the 34th Intl Symp on Computer Architecture*, Jun. 2007.

[9] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood, "TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory," in *Procs. of the 35th Intl Symp on Computer Architecture*, Jun. 2008.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," in *Procs. of the 12th Intl Conference on Architectural Support for Programming Languages and Operating Systems*.

[11] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," in *Procs. of the ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, Mar. 2006.

[12] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Procs. of the 34th Intl Symp on Computer Architecture*, Jun. 2007.

[13] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation," in *Procs. of the 14th Intl Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.

[14] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Procs. of the 13th Intl Symp on High-Performance Computer Architecture*, Feb. 2007.

[15] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "The Common Case Transactional Behavior of Multithreaded Programs," in *Procs. of the 12th Intl Symp on High-Performance Computer Architecture*, Feb. 2006.

[16] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Procs. of The IEEE Intl Symp on Workload Characterization*, Sep. 2008.

[17] M. Lupon, G. Magklis, and A. Gonzalez, "Version Management Alternatives for Hardware Transactional Memory," in *Procs. of the 9th Workshop on Memory performance: dealing with applications, systems and architecture*, Oct. 2008.

[18] J. R. Titos, M. E. Acacio, and J. M. Garcia, "Characterization of Conflicts in Log-Based Transactional Memory," in *Procs. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb. 2008.

[19] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A Scalable Approach to Thread-Level Speculation," in *Procs. of the 27th Intl Symp on Computer Architecture*, Jun. 2000.

[20] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Woo, "OS Support for Virtualizing Hardware Transactional Memory," in *Procs. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[22] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Procs. of the 22nd Intl Symp on Computer Architecture*, Jun. 1995.

[24] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "MetaTM/TxLinux: Transactional Memory for an Operating System," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, 2007.

[25] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott, "An Integrated Hardware-Software Approach To Flexible Transactional Memory," in *Procs. of the 34th Intl Symp on Computer Architecture*, Jun. 2007.

[26] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Procs. of the 33th Intl Symp on Computer Architecture*, Jun. 2006.