A Selective Logging Mechanism for Hardware Transactional Memory Systems

Marc Lupon*
mlupon@ac.upc.edu

Grigorios Magklis†
grigorios.magklis@intel.com

Antonio González*†
antonio.gonzalez@intel.com

* *Computer Architecture Department, Universitat Politècnica de Catalunya*
† *Intel Barcelona Research Center, Intel Labs Barcelona - UPC*

***Abstract*—Log-based Hardware Transactional Memory (HTM) systems offer an elegant solution to handle speculative data that overflow transactional L1 caches. By keeping the pre-transactional values on a software-resident log, speculative values can be safely moved across the memory hierarchy, without requiring expensive searches on L1 misses or commits. Unfortunately, software logging incurs significant overheads that may affect the performance of applications with large transactions.**

In this paper, we present *selective logging*, a novel mechanism that adds to the software-resident log only the speculatively modified lines that overflow the transactional L1 cache. In particular, we present how two distinct HTM systems can combine selective logging with built-in transactional caches to improve on their performance. Our evaluation shows that selective logging reduces the latency of transactional stores, speeds up the abort recovery of long transactions and increases the utilization of transactional caches. What is more, our studies show that selective logging provides more flexibility, as it supports different conflict management strategies.

***Keywords*-Selective logging, log-based HTM systems, FASTM-SL, SPECTM**

I. Introduction

Data version management (VM) is one of the key aspects of Hardware Transactional Memory (HTM) systems that employ speculation to execute transactions [12]. This mechanism defines *how* and *where* the values generated within a transaction are kept, as well as *what* actions must be satisfied at commit and abort time. Thus, the VM strategy followed when implementing an HTM system directly impacts the performance of transactional applications and the complexity of the hardware design [4].

A high-performance HTM system must accelerate transactions of any size or duration. To this end, unbounded HTM systems incorporate hardware support to maintain overflowing data on the side. Log-based HTM systems—those that store transactional modifications in-place in memory while they keep pre-transactional values on a software-resident log [20]—are widely accepted as one of the best VM alternatives to handle the transactional state because they strike a delicate balance between complexity, hardware cost and performance [2], [3], [13], [27].

Conventional logging mechanisms face three main challenges that may slow down transactional execution. First, writing pre-transactional values to the log always before a transactional store enlarges the latency of transactional stores—speculative values cannot be written in memory until the old data is logged. Second, the software log is maintained in cacheable memory, which reduces the buffering capacity of the transactional caches—in other words, the logging mechanism increases the possibilities of evicting transactional data. Third, in log-based HTM systems *all* transactionally written lines are placed in the log, even if the speculative data fits in the L1 cache. Thus, if an overflowing transaction aborts, it has to restore the whole log using a *slow* software routine, ignoring the built-in hardware support of the transactional caches [1], [3], [16], [22].

What is more important, log-based data versioning enforces *eager* conflict management (*i.e.*, conflicts have to be resolved at the moment that they are produced), to guarantee that only a single copy of a transactionally modified line is alive in the system. Nonetheless, previous studies showed that more aggressive policies like *lazy* conflict management (*i.e.*, memory inconsistencies are resolved when transactions reach their end) obtain better performance results [24], [17]. Unfortunately, these policies are incompatible with conventional logging mechanisms.

In order to address the above issues, this paper proposes *selective logging*, a novel VM technique that only logs the pre-transactional values of those memory blocks that the hardware cannot recover—*e.g.*, a non-committed speculative write that overflows transactional buffers. Similar to other log-based HTM systems, evicted speculative data is stored in-place in the shared levels of the memory hierarchy; therefore our proposal does not produce delays on cache misses or commits.

The idea behind selective logging is rather simple but effective. By adding a few additional hardware steps on resource overflows (uncommon event), we are able to (i) accelerate most of the memory updates within a transaction, (ii) reduce the size of the software log, which accelerates the abort recovery process (fewer lines must be restored by software) and (iii) provide flexible con-

flict management for non-overflowing memory blocks. Our evaluation shows that selective logging obtains an average speed-up of 36% compared to modern log-based HTM systems when executing workloads that commonly overflow transactional buffers.

The main contributions of this work are twofold. First, we quantify the limitations of conventional logging in well-known HTM systems. Second, we integrate selective logging into two distinct HTM systems. In particular, our approach is the first to offer deferred resolution of conflicts in a log-based HTM framework.

The remainder of the paper is organized as follows. In Section II, we summarize related work on unbounded HTM, with special focus on the VM mechanism for overflowing data. In Section III, we describe how the selective logging mechanism works and discuss the hardware implementation. In Section IV, we present FASTM-SL, a case for an eager HTM with selective logging, while in Section V we detail SPECTM, a case for a lazy HTM that supports *early* memory updates when speculative data exceeds transactional buffers. In Section VI we evaluate our proposals and in Section VII we conclude the article.

II. Background in VM mechanisms

Early HTM assumed finite hardware support (private transactional caches [11] or local store buffers [21]) to hold transactional modifications, which disallowed the execution of transactions that exceeded the buffering support. A simple way to permit such transactions is falling back to a Software Transactional Memory (STM [23]) system when buffers are overflowed, which is commonly known as Hybrid TM (HyTM) system [10], [14], [15]. Unfortunately, STMs still incur significant overheads [6] that considerably downgrade the performance of applications with many large transactions.

To speed up the execution of any kind of transaction, several HTM systems extend finite transactional buffers with additional hardware support—we refer to these approaches as unbounded HTM systems. These HTM proposals fall into one of two distinct design strategies while implementing VM for overflowing data: they are either *deferred update* (also known as *lazy* VM) or *log-based* (also known as *early update* or *eager* VM) HTM systems.

Deferred update HTM systems [22] keep overflowed data hidden from in-flight transactions using specialized structures, such as firmware-accessed memory structures (LTM [1] or VTM [22]), shadow memory pages (PTM [7] or XTM [8]) or additional hardware tables (FlexTM [25] or EazyHTM [26]). When the transactional buffers are overflowed, the system inserts new data in these specialized structures, where it is kept until the transaction commits (new data is transferred to global memory) or aborts (new data is invalidated). Also, if the transaction does not find the data in the transactional buffers, deferred update HTM systems must traverse the specialized structures to check if the accessed data has been modified during the in-flight transaction. Hence, these HTM systems are subjected to long delays when they execute transactions that commonly exceed on-chip data versioning support.

On the other hand, log-based HTM systems [20] keep new state in-place in memory, holding the pre-transactional data in a software-accessed log that contains the old values of transactionally modified lines and their associated addresses [2], [3], [27]. In the case of abort, the system must trigger an exception and recover pre-transactional values using a user-level (slow) software routine. Nonetheless, commits are immediate—data is already placed in memory. Thus, log-based HTM systems do not suffer from resource overflow like deferred update HTM systems, as speculative data can be safely moved across the memory hierarchy.

III. The Selective Logging Mechanism

For our selective logging implementation, we assume a Chip Multiprocessor (CMP) with single-threaded cores and two levels of caches: a private, transactional L1 cache that tracks transactionally modified data using new coherence states [1], [3], [16], [22] and a distributed, shared L2 cache that keeps pre-transactional values of non-replaced data. Coherency is implemented using a directory placed close to the L2 cache.

In HTM systems with selective logging, transactional stores do not carry additional actions—*i.e.*, it is not necessary to write in the log the old state before updating the memory. However, when a transactional line is evicted from the L1 cache, the processor stops conventional execution (the memory instruction that generates the cache miss remains incomplete) and starts executing a microcode routine that loads the old value of the line from the L2 cache into a special register, and stores the old data and the corresponding memory address in the first free entry of the software log. After that, the processor re-schedules the memory instruction that produced the cache replacement and continues executing the transaction.

Like other log-based HTM systems, commits do not require additional actions, given that the transactional state can harmlessly flow though the memory hierarchy. Nonetheless, when an overflowing transaction aborts it has to perform a two-phase procedure. First, the hardware invalidates all the transactional lines in the L1 cache, clearing the transactional state from caches. Then, the processor throws an exception and traps to the user (or system) software layer, which undoes the modifications introduced by the transaction.

A. Pushing Memory Addresses in the Log

By deferring log updates to L1 eviction time, selective logging requires a subtle modification to the way the log is stored in memory. More specifically, traditional log-based HTM systems use *logical* addresses to track the location of pre-transactional data. Logical addresses are readily available at the transactional store issue time (*i.e.*, the time the log info is collected). The benefit of using logical addresses in the log is that the software recovery routine can be done in user-space.

In selective logging, on the other hand, the system collects the log info at the time a L1 cache line is evicted. At this point, logical addresses are not available (most memory systems use physical addresses), but using physical addresses in the log poses a security risk though.

In order to address the above issues, we propose to move the transaction abort recovery handler in the Operating System (OS). In this case, when an overflowing transaction aborts, the hardware raises an exception that calls the OS abort recovery routine. The OS recovers the log, using physical addresses, and returns control to the application. Note that logical-to-physical translation is not needed when the OS is undoing the log—the TLB is automatically bypassed.

Moreover, the actual log memory must be only visible to the OS, otherwise user applications can reverse-engineer the logical-to-physical memory mapping. This requires that transactional applications execute a log creation system call at init time. The memory of the log is thus kept in OS memory, and is hidden from the application.

In the next sections, we show how selective logging can be efficiently implemented in two different HTM environments. For those case studies, we assume that the log contains physical addresses, and thus it has to be recovered in privileged mode.

IV. A Case for Speculative Logging in Eager HTM Systems

In this section, first we describe FasTM, that we use as our starting point for an *eager* HTM system [16], and then we couple FasTM with selective logging to further improve its performance.

A. The Base FasTM System

FasTM is a log-based HTM system that combines early memory updates for overflowing data with deferred memory updates for transactions that fit in the L1 cache. In FasTM, transactional updates are stored in the L1 cache in a special state (called `T`), until the transaction commits, aborts, or until the cache line is evicted. The system must guarantee that before transitioning a line to the `T` state, the pre-transactional value of the line is written back to the L2 cache (by forcing a write-back). This guarantees that the memory hierarchy always holds the correct pre-transactional data for transactionally modified lines that are not evicted from the L1 cache.

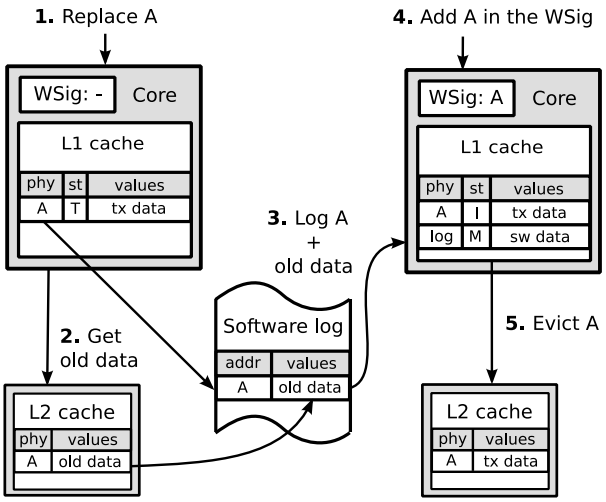


Figure 1. L1 Cache replacement actions in FASTM-SL

When a transaction aborts, it checks if any `T` lines have been evicted. If not, all transactionally written lines in the L1 cache are instantly invalidated, enabling a very fast abort recovery. For overflowing transactions, FasTM walks the log using a software routine, similar to LogTM [20]. Similar to LogTM-SE [27], FasTM introduces `Read` and `Write` Signatures to maintain a superset of the memory locations accessed within a transaction.

In this section, we describe how we extend the FasTM infrastructure with selective logging—we call this system FasTM-SL. FasTM-SL differs from FasTM in the way it updates the software log and how it recovers the pre-transactional state when aborting an overflowing transaction. While FasTM logs the values of *all* transactional stores (at least the first time they write a line inside a transaction), FasTM-SL only logs the values of transactional evicted data. Thus, if an overflowing transaction aborts, FasTM has to restore the entire pre-transactional state by software. Instead, FasTM-SL can take advantage from the innate in-cache support for clearing non-evicted cache lines.

B. L1 Cache Evictions

When a transactionally written line is evicted from the L1 cache, FasTM-SL has to construct a new log entry. We use the example of Figure 1 to describe how the selective logging machinery handles the eviction of a `T`-state line (step 1). First, the eviction process is put on hold, and the core sends a request to the L2 cache for the previous version of the line (step 2).

The requested data, together with the physical address of the line are temporarily stored in a special register. At this point, the data in the special register is written to the first free entry of the log using regular (*i.e.*, non-transactional) memory operations (step 3), and the physical address of the line is added to the `Write` Signature (step 4). Finally, the transactional line is evicted to the L2 cache.

C. Hardware/Software Abort Recovery

When an overflowed transaction aborts, FASTM-SL has to restore the values modified during its execution. For non-evicted data it is enough to invalidate `T`-state lines (by flash-clearing the state bits), as pre-transactional values are still valid in the L2 cache. However, transactional replaced data has to be restored by software because the L2 cache does not hold the old state anymore. Hence, the system triggers an exception, which jumps to an Operating System routine that walks the log in reverse order to undo the changes introduced by the aborted transaction.

Note that, in contrast to FASTM, FASTM-SL *only* has to restore those lines that have been evicted from the L1 cache during the in-flight transaction—those lines that fit in the L1 cache are invalidated by the underlying hardware and, eventually, the core will obtain the valid data from the L2 cache using conventional coherence requests when the transaction restarts. As a result, the size of the log (and thus the time spent in software abort recovery) is reduced considerably.

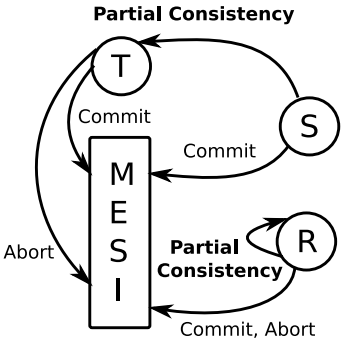
V. A CASE FOR SPECULATIVE LOGGING IN LAZY HTM SYSTEMS

To the best of our knowledge, all HTM systems that postpone the resolution of conflicts (*e.g.*, lazy conflict management) implement *deferred update* VM to store the overflowing speculative state [11], [8], [26]. Instead, our approach (we call it SPECTM) takes an opposite direction when dealing with speculative data that exceed the size of the L1 cache: our system implements *early update* VM on overflowing data, moving pre-transactional data to the software log and placing new data on the shared levels of the memory hierarchy.

A. The Base SPECTM System

SPECTM assumes a simplification of the UTCP coherence protocol implemented in [17], which allows multiple versions of the same memory line in distinct L1 caches. Like FASTM, non-conflicting transactional updates are held in the `T` state. However, conflicting lines are kept in two separate states: `R` for reads and `W` for writes. Pre-transactional values of non-evicted cache lines are always kept in the L2 cache.

Similar to FlexTM [25] or EazyHTM [26], cores use a conflict list (`CL`) to track those transactions with whom they have a conflict with. Before committing, transactions must request the abort of those transactions that are present in the `CL`. If the `CL` is empty, the commit process starts immediately. After that, the transactional state becomes globally visible—`T` and `W` lines transit to Modified, and `R` lines transit to Invalid.



State Codification

State	S	T	D	V
W	1	1	1	0
T	0	1	1	0
R	0	1	0	0
M	0	0	1	0
E	0	0	1	1
S	0	0	0	1
I	0	0	0	0

Figure 2. Partial consistency transitions on the UTCP protocol

Using *early updates* on overflowing data presents non-trivial challenges for SPECTM. First, HTM systems with lazy conflict management allow multiple versions of a line in distinct L1 caches. However, before replacing a cache line, the system must guarantee that the evicting core is the unique owner of that line, because that core is responsible of restoring its old state if the transaction aborts. Thus, an overflowing line must only have a single copy in the system.

Second, our approach moves overflowing data to the shared memory space, overwriting the old state kept in the L2 cache. As the old value of the line can no longer be obtained, the system must prevent remote transactions to access transactionally evicted lines, preserving those lines isolated from the world until the transaction commits or aborts.

To achieve the above goals, SPECTM adds two novel mechanisms: *Partial Consistency* and *Overflow Isolation*. The next sections describe how these mechanisms operate as well as how they are implemented.

B. Partial Consistency

Before writing back the value of the transactionally modified cache line, the system must ensure that there are no live copies of the line in other private L1 caches. This is a straightforward step for consistent `T`-state lines, as they are exclusively owned by a single core. However, conflicting written lines—those lines that have been moved to the `W` state and thus potentially have multiple readers/writers in non-committed transactions—require additional actions to eliminate non-compatible values.

In order to invalidate all the transactional sharers of the cache line, the evicting core sends an *Abort* notification to all the cores that are present in its `CL`,

following exactly the same procedure as at commit time. When the remote cores receive that message, they abort, even if they have not touched the evicted cache line within their transactions. If an aborting transaction has already overflowed the L1 cache—*i.e.*, the software log is not empty—then the core must recover the old state using the procedure described in Section IV-C.

After the abort process ends successfully, the evicting core is safe to write back the speculative value in the L2 cache. However, before that, the core (i) transits all the `W`-state lines to `T`, (ii) clears the `CL` list and (iii) inserts the memory address of the evicted cache line in the `Write` Signature.

Partial Consistency guarantees that the replaced cache line has a unique owner in the system, as potential conflicters have aborted. What is more, it also guarantees that, at that point, *every* line being written inside the transaction only belongs to the overflowing transaction. Nonetheless, that transaction is not entirely isolated from the rest, as conflicting read lines—those kept in the *R* state—may still be found in the write set of other in-flight transactions. Figure 2 shows how Partial Consistency modifies the transitions of the UTCP Coherence Protocol.

C. Overflow Isolation

In SPECTM, overflowing data must be preserved in isolation—no in-flight transaction can access that data until the transaction commits or aborts. To guarantee that invariant, when a core receives a coherence request from a remote transaction, it checks its `Write` Signature. If the address is present in the filter, then the core replies with an *Abort* message, and the remote transaction aborts immediately.

Note that overflowing transactions may produce cascades of aborts (either when there are continuous evictions of transactional data or when overflowing transactions access speculative data that has been moved out of the L1 cache). As these transactions have to be recovered by software, we decided to perform a randomized exponential backoff before restarting the transaction. This strategy reduces contention to ensure forward progress in the application.

For exemplifying how overflowing data is kept in isolation, we assume a transaction `Ti` that has written lines *A* and *B*, and a transaction `Tj` that has accessed line *B*. Eventually, `Ti` replaces line *A* from the L1 cache, causing the abort of transaction `Tj`. At that moment, all transactionally written lines by transaction `Ti` are consistent, including blocks *A* and *B*. After that, we assume that another transaction `Tk` attempts to write the evicted line *A*. However, the system denies the access to preserve the isolation of the overflowed cache block, aborting transaction `Tk`.

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
Directory	Bit vector of sharers/owners, 6-cycle latency
Interconnect	16-node Mesh, 64-byte links, 2-cycle wire latency, 1-cycle router latency
Coherency	Unified Transactional Coherence Protocol
Signatures	2 Kb Parallel Cuckoo Bloom filters

Table I
BASE SYSTEM PARAMETERS

VI. EVALUATION

For our evaluation of selective logging, we assume a Chip Multiprocessor (CMP) with 16 cores and two levels of caches, where the first level (L1) is private and the second level (L2) is shared among all the cores. Coherency is implemented using a blocking, distributed directory placed in the L2 cache. The system has a 16-node mesh interconnect that uses 64-byte links with adaptive routing. Each node has a core, a piece of a shared L2 cache and part of the directory. The system has four memory controllers to access 4 GB of main memory. Each core tracks transactional memory accesses in two 2 Kbit `Read` and `Write` Signatures. Detailed system parameters are shown in Table I.

A complete HTM system has been simulated using the Simics [18] infrastructure from Virtutech and the GEMS [19] toolset from Wisconsin's Multifacet group. For our performance analysis, we have chosen a set of applications from the STAMP benchmark suite [5] and three in-house benchmarks that perform atomic operations in huge data structures. We have selected those applications because they typically execute large transactions that overflow the L1 cache, and thus they are more sensitive to the VM strategy implemented in the base HTM system.

Table II provides detailed information about the applications we utilize. The first three columns show the benchmark suite, the name of the application, and its input parameters. The fourth column (Commit) shows the number of committed transactions on the application, and the next two columns show the average size of the read set (Rd Set) and the write set (Wr Set). These numbers were collected running FASTM with 16 threads.

A. Base HTM Systems

For our performance analysis we have chosen to compare the HTM systems with selective logging (FASTM-SL as an eager HTM system, SPECTM as a lazy HTM system) with two other systems that use conventional logging (FASTM [16] as an eager HTM system, DYNTM [17] as a lazy HTM system).

						FASTM			FASTM-SL		
Suite	Bench	Input parameters	Commit	Rd Set	Wr Set	OV	Aborts	SW Ab	OV	Aborts	SW Ab
μbench	Btree	50/50 ins/look, 32 op/tx	2048	155.9	88.1	627	0.18	6.5%	34	0.18	2.2%
	Lists	16 lists, 2K updates	8192	30.4	30.2	802	0.53	7.7%	9	0.42	0%
	Hash	25/50/25 ins/look/del	4096	114.5	104.7	1814	0.67	12.6%	830	0.63	4.6%
STAMP	Bayes	32 vars, 1024 records	520	82.6	41.4	71	3.4	16%	37	2.3	10%
	Labyrinth	32*32*3 maze, 2K routes	4128	111.8	101.6	2318	0.22	57.9%	379	0.8	1%
	Vacation	64K c, 80% q, 16 items	16384	143.1	22.4	2361	0.10	11%	2080	0.16	9%
	Yada	20 angle, 633.2 mesh	2966	32.2	14.3	511	2.01	2%	24	2.0	0.6%

Table II
INPUT PARAMETERS AND CHARACTERIZATION OF TRANSACTIONAL APPLICATIONS

For FASTM and FASTM-SL we implemented the **Stall** conflict resolution policy [27], which stops the conflicting requester until the violation disappears. To eliminate deadlocks among stalled transactions, the system uses a timestamp to abort the younger transaction that participates in a cycle. After recovery, a randomized exponential backoff is performed to avoid livelocks. For DYNTM, we remove the predictor from the core to provide a fair comparison with SPECTM, a fixed lazy HTM system that cannot dynamically adapt its execution. Hence, this DYNTM implementation executes all the transactions that fit in the L1 in lazy mode. If a transactionally written line leaves the L1, the system aborts the transaction and restarts it in eager mode, which is similar to FASTM .

We also compare our proposal with two (eager and lazy) idealized VM implementations that serve as upper-bounds. These implementations never log values in software; instead they keep transactionally evicted lines in an infinite victim cache. This cache has the same latency as the L1 for reads and writes. The transactional victim cache moves committed values to the L1 instantaneously, and it has a zero-cost abort recovery—transactional entries are just discarded.

B. FASTM-SL Performance Analysis

Figure 3 presents the time distribution of FASTM (labeled F), FASTM-SL (labeled S) and Ideal Eager (labeled E) HTM systems in their 16-threaded executions. The execution time has been normalized to the 16-threaded FASTM execution and is broken down to: non-transactional and barrier cycles (labeled Non-Tx and Barrier), the time spent in committed (labeled Good Tx) and aborted (labeled Aborted Tx) transactions, the time consumed 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). The number on top of each bar shows the speed-up achieved over a single-threaded FASTM execution.

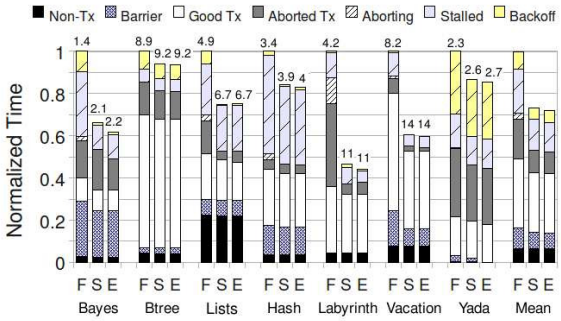


Figure 3. Normalized execution time of eager HTM systems

As it can be seen in Figure 3, FASTM-SL obtains a 36% speed-up over FASTM (27% reduction of execution time), obtaining similar performance to the Ideal Eager approach. The benefit is especially noticeable in *Labyrinth* or *Vacation*, which achieve almost 2X speed-up. The reasons for this behavior are the following.

Small log size. Selective logging drastically reduces the number of cache lines that have to be maintained in software. Figure 4 shows the average size (in KB) of the software log per transaction in FASTM and in FASTM-SL. Selective logging drastically lowers the size of the log by a factor of 15X (in *Hash* almost a 100X). This fact has two implications. First, there are less transactions that overflow the L1 (Table II, labeled OV). Second, as there is more space in the L1 for caching transactional data, the hit rate of the L1 increases higher.

Efficient transactional stores. In FASTM-SL, transactional stores do not need to access the software log each time they are retired—only when they leave the L1, which is an uncommon event. As a result, the time spent in transactions that commit (Good Tx in Figure 3) is reduced by 14% on average.

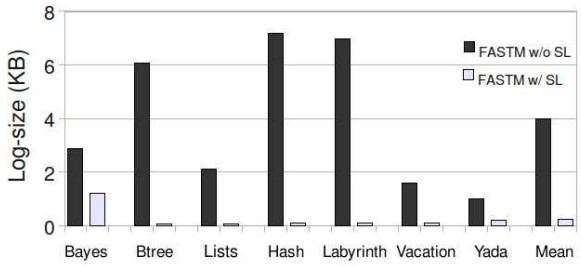


Figure 4. Software log size in FASTM and FASTM-SL

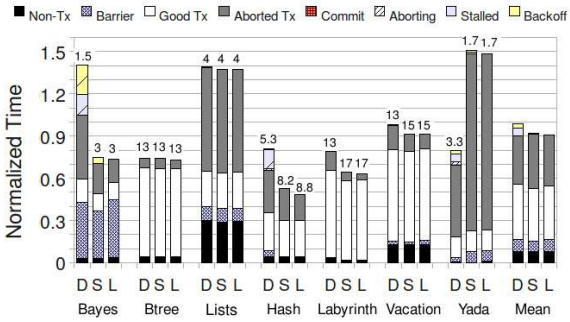


Figure 5. Normalized execution time of lazy HTM systems

Fast abort recovery. In case of abort, the software has to restore just a few lines. Moreover, the number of software aborts is also reduced in FASTM-SL because less transactions overflow the L1 (Table II, labeled SW Ab). Accordingly, Figure 3 shows that FASTM-SL virtually eliminates the abort recovery overhead. As pointed out in previous studies [16], speeding up aborts cuts down the time that transactions are exposed to conflicts, which turns out to lower the abort rate (Table II, labeled Aborts) and the time spent in Stall and Backoff cycles.

Notice that this evaluation only shows the results of applications with coarse-grained transactions. Applications with small (non-overflowing) transactions from STAMP (*e.g.*, *Kmeans* or *Ssca2*) or from the SPLASH2 benchmark suite only report speed-ups between 1% to 3%, but never perform worse in FASTM-SL.

C. SPECTM Performance Analysis

Figure 5 presents the time distribution of DYNTM (labeled D), SPECTM (labeled S) and Ideal Lazy (labeled L) HTM systems in their 16-threaded executions. The execution time has been normalized to the 16-threaded FASTM-SL execution and is broken down using the same criterion from Figure 3, plus the time spent in committing transactions (labeled Commit).

As it is shown in Figure 5, SPECTM outperforms FASTM-SL in 5 of the 7 applications, all except *Lists* and *Yada*. (FASTM-SL is the baseline in Figure 5, where its execution time is normalized to 1). This is because lazy HTM systems permit more concurrency than eager HTM systems when transactions collide, eliminate some read-write violations and do not require backoff. For instance, *Bayes*, *Btree* and *Labyrinth* reduce FASTM-SL execution time up to a 30%. Note that *Labyrinth* achieves 17X speed-up over single-threaded FASTM execution when it uses 16 threads because (i) the application scales pretty well in lazy mode and (ii) SPECTM does not log values on every transactional store, whereas FASTM has to do it.

SPECTM obtains close performance to the Ideal Lazy HTM system. Nonetheless, DYNTM cannot achieve such benefits given that it has to fallback to eager mode on overflows, the same way that some HyTMs [9], [14] switch to STM execution to handle large transactions. This results to a significant amount of discarded work and prevents the use of lazy conflict management on large transactions. Instead, SPECTM continues execution through overflows and guarantees deferred conflict resolution for the majority of lines—those that fit in the L1. In applications like *Bayes* or *Hash*, SPECTM achieves up to 60% speed-up over DYNTM. On the other hand, DYNTM can take advantage of using eager conflict management on large transactions in *Yada* to improve on FASTM-SL. On average, SPECTM produces a 7% speed-up over DYNTM .

VII. CONCLUSIONS

Log-based HTM systems offer an elegant solution to handle unbounded transactions. Unfortunately, conventional logging mechanisms introduce significant overheads such as delays on transactional stores, higher L1 miss rates, or slow aborts on overflowing transactions. Selective logging proposes a novel approach for HTM systems that implement early updates. Our approach moves to the software-resident log only those old values that are not maintained in hardware, permitting a more effective utilization of transactional resources.

We present two implementations that use selective logging: FASTM-SL as an eager HTM system and SPECTM as a lazy HTM system. We have evaluated both approaches with overflow-sensitive transactional applications. FASTM-SL and SPECTM obtain, on average, a speed-up of 36% and 7% over two modern HTM systems, achieving similar performance than idealized HTM systems. We have seen that implementing selective logging accelerates transactional execution, reduces the number of slow aborts and decrements the size of the software log. What is more, selective logging opens new avenues on systems that enforce deferred conflict management by permitting early updates on overflowing data.

REFERENCES

[1] 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*, February 2005.

[2] 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*, June 2007.

[3] 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*, June 2008.

[4] 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*, June 2007.

[5] 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*, September 2008.

[6] C. Cascaval, C. Blundell, M. Micheal, H. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why is it only a research toy?" *Communications of the ACM*, vol. 51(11), pp. 40–46, November 2008.

[7] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded Page-Based Transactional Memory," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[8] J. Chung, C. Cao Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "Tradeoffs in Transactional Memory Virtualization," in *Procs. of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," in *Procs. of the 12th Intl Conf on Architectural Support for Programming Languages and Operating Systems*, October 2006.

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

[11] 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*, June 2004.

[12] 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.

[13] S. Jafri, M. Thottethodi, and T. Vijaykumar, "LiteTM: Reducing Transactional State Overhead," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan. 2010.

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

[15] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased Transactional Memory," in *Procs. of the 2nd Workshop on Transactional Computing*, August 2007.

[16] M. Lupon, G. Magklis, and A. González, "FASTM: A log-based hardware transactional memory with fast abort recovery," in *Procs. of the 18th Intl Conf on Parallel Architectures and Compilation Techniques*, September 2009.

[17] M. Lupon, G. Magklis, and A. Gonzalez, "A Dynamically Adaptable Hardware Transactional Memory," in *Procs of the 43rd Intl Symp on Microarchitecture*, December 2010.

[18] 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, 2002.

[19] 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, 2005.

[20] 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*, February 2006.

[21] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *Procs. of the 34th Intl Symp on Microarchitecture*, December 2001.

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

[23] N. Shavit and D. Touitou, "Software Transactional Memory," in *Procs. of the 14th Symp on Principles of Distributed Computing*, August 1995.

[24] A. Shriraman and S. Dwarkadas, "Refereeing Conflicts in Hardware Transactional Memory," in *Procs. of the 23rd Intl Conference on Supercomputing*, June 2009.

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

[26] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazy-HTM, Eager-Lazy Hardware Transactional Memory," in *Procs. of the 42nd Intl Symp on Microarchitecture*, December 2009.

[27] 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*, February 2007.