Version Management Alternatives for Hardware Transactional Memory*

Marc Lupon
Dept. d'Arquitectura de Computadors
Univ. 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

Transactional Memory is a promising parallel programming model that addresses the programmability issues of lock-based applications using mechanisms that are transparent to developers. Hardware Transactional Memory (HTM) implements these mechanisms in silicon to obtain better results than fine-grain locking solutions. One of these mechanisms is data version management, that decides how and where the modifications introduced by transactions are stored to guarantee their atomicity and durability.

In this paper, we show that aborts are frequent especially for applications with coarse-grain transactions and many threads, and that this severely restricts the scalability of log-based HTMs. To address this issue, we propose the use of a gated store buffer to accelerate eager version management for log-based HTM. Moreover, we propose a novel design, where the store buffer is used to perform lazy version management (similar to Rock [12]) but overflowed transactions execute with a fallback log-based HTM that uses eager version management.

Assuming an infinite store buffer, we show that lazy version management is better suited to applications with fine-grain transactions while eager version management is better suited to applications with coarse-grain transactions. Limiting the buffer size to 32 entries, we obtain 20.1% average improvement over log-based HTM for applications with fine-grain transactions (using lazy version management) and 54.7% for applications with coarse-grain transactions (using eager version management).

Categories and Subject Descriptors

B.1.3 [**Memory Structures**]: Design Styles—*Shared Memory*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Performance, Design, Languages

Keywords

Hardware Transactional Memory, Version Management

*This work is supported by the Spanish Ministry of Education and Science and FEDER funds of the EU under contract TIN2007-61763, the Generalitat de Catalunya under grant 2005SGR00950, and Intel Corporation

1. INTRODUCTION

Transactional Memory (TM) [8] provides an alternative, lock-free, parallel programming model that provides non-blocking synchronization among transactions. A transaction is a sequence of reads and writes on shared memory data that can be safely executed in parallel with the rest of the program. Similar to database transactions, TM defines the execution of a transaction to be atomic, isolated, durable and consistent. Hardware Transaction Memory (HTM) [1, 7, 8, 14] proposes to utilize specialized hardware to accelerate the underlying mechanisms of transactional execution.

Previous studies [6] claimed that common-case transactions were short and did not usually conflict, suggesting simple recovery mechanisms to resolve conflicts [14, 21]. However, more recent transactional workloads [5] have introduced large transactions that access different data structures concurrently, which generates contention and produces unwanted overhead.

One of the underlying mechanisms of TM is version management. Version management defines (1) where (and how) are the transactional and pre-transactional states stored, (2) how the state is updated at commit, and (3) how the state is updated at abort and recovery. There are two major policies and each can be implemented using different techniques [4]. In this paper we analyze the impact of version management policies in an HTM environment.

Lazy version management does not make memory updates from a transaction visible to other threads. This is done either by storing the new values in different memory locations or by using special buffering to temporarily hold the new data. On commit, the new values must be made visible to the other threads. Eager version management on the other hand makes updates visible immediately (but it must maintain the pre-transaction values on the side). Commit is immediate, but abort requires to restore the old values.

First, we characterize LogTM-SE [21], a log-based HTM system that uses eager version management. LogTM-SE as well as other recently proposed log-based systems [2, 3] use hardware to accelerate conflict detection, but perform abort recovery by software (to be able to support very big transactions). Our characterization is similar to that of Titos *et al.* [19], but we use a wider spectrum of benchmarks and configurations with more cores. Our analysis shows that abort recovery can lead to serious performance opportunity loss for applications with coarse-grain transactions.

This motivates the use of specialized hardware to accelerate version management. We have decided to use a gated store buffer similar to Crusoe™ [9] and Rock [12]. Fur-

thermore, we propose to use the buffer to implement both eager and lazy version management. For lazy version management, transactional stores are kept in the buffer, and are pushed to the memory hierarchy at commit. For eager version management, pre-transactional data are put in the buffer before transactional stores overwrite their memory locations, and are pushed to the memory hierarchy at abort.

We perform two evaluations of this design: the first assumes infinite buffers and the second is a realistic implementation. Comparing LogTM-SE with the idealized eager approach we measure the opportunity loss of unaccelerated log-based HTM. We also analyze the differences between the idealized eager and lazy approaches. We conclude that a lazy policy is better suited to applications with fine-grain transactions while an eager policy is better suited to applications with coarse-grain transactions.

The issue with finite hardware is that large transactions may overflow the buffering space. For both eager and lazy version management we propose to utilize LogTM-SE to handle transactions that overflow the buffer. The use of LogTM-SE significantly simplifies previous proposals that require complex structures to execute overflowed transactions [1, 15] while outperforming classical log-based proposals that use software-only recovery mechanisms [2, 21]. Utilizing a 32-entry store buffer, we obtain 20.1% average improvement over LogTM-SE for applications with fine-grain transactions (using lazy version management) and 54.7% for applications with coarse-grain transactions (using eager version management).

The contributions of this paper are three-fold. First, our characterization of LogTM-SE expands on previous work with new benchmarks. Second, our idealized eager vs. idealized lazy version management analysis with an extensive list of applications draws interesting conclusions about an area largely neglected in TM studies. Third, our two proposed implementations of hardware-accelerated version management (eager and lazy) with a log-based HTM fallback.

The rest of the paper is organized as follows: Section 2 summarizes data version management mechanisms in current HTMs. Section 3 presents in detail a baseline log-based HTM. Section 4 discusses how the inclusion of a store buffer can accelerate eager and lazy version management mechanisms and explains the implementation of our system in the face of buffer overflows. Section 5 evaluates the various version management techniques presented with both infinite and finite resources, and Section 6 concludes this study.

2. RELATED WORK

Herlihy and Moss [8] introduced HTM as a new multiprocessor architecture intended to make lock-free mechanisms as efficient as conventional techniques based on mutual exclusion. Their design uses traditional cache management and coherence on non-transactional operations, and provides extra instructions for transactional accesses, commit actions and state validation. Modifications introduced by a transaction were tracked in a separate processor cache that contained old and new values, which could only be accessed by the owner processor.

Transactional Coherence and Consistency (TCC) [7] proposed a consistency model based on transactions. TCC uses lazy version management: the first level caches buffer new values locally, while a second level shared cache holds the old values. Transactions send all their modifications to the second level cache at commit time making the changes visible to all processors.

Original HTM proposals posed limitations on the size or duration of transactions. Recently, there are several proposals able to execute unbounded transactions using finite hardware. Hybrid Transactional Memory (HyTM) [10] uses Software Transactional Memory (STM) [16] to handle large transactions, whereas common-case, smaller transactions use best-effort hardware. LTM [1] and VTM [15] are lazy, unbounded HTMs that spill transactional state into an overflow data structure in main memory. This technique is quite expensive, because, if a requested line has been updated during the transaction, the system must walk the overflow structure to get the right value.

RTM [17] performs lazy data version management by using the buffer capabilities of private caches. RTM adds two states (TI for reads and TMI for writes) to the MESI protocol to track the lines used by transactions and to ensure isolation and consistency. On commit, lines in the TMI state change to Modified, and on abort lines in TMI and TI states change to Invalid.

Rock [12] will possibly be the first processor to include transactional hardware support. RockTM utilizes lazy data version management, using a finite buffer that stores transactional writes at word granularity. The buffered stores become visible to the memory system when a transaction commits. In case of overflow, the system triggers an exception to re-execute the overflowed transaction using software mechanisms.

LogTM [14] performs eager data version management by storing old values and their associated addresses in a private software log. Read-Write cache bits are used to detect conflicts, whereas new "sticky" directory states maintain concistency for evicted transactional lines. LogTM-SE [21] decouples transactional state from caches, replacing Read-Write bits with signatures. A signature is a compact representation of a set of memory addresses. A signature supports conflict detection efficently, but it is inexact (we may get false positives).

Like LogTM-SE, OneTM [2] and TokenTM [3] use log-based eager data version management, but implement different strategies to track the locations accessed by transactions. OneTM introduces a permissions-only cache to maintain consistency among transactions that evict lines from private buffers, but only allows one transaction to overflow in this cache at a time. TokenTM adapts the concept of token coherence to detect conflicts among transactions and eliminate false positives produced by finite signatures.

3. LOG-BASED HTM

We have chosen LogTM-SE [21] as our log-based baseline system for its simplicity and the large amount of literature that it has generated. LogTM-SE is a log-based HTM with eager version management that utilizes a software handler to restore transactional state in case of abort.

LogTM-SE uses signatures to track the locations accessed by transactions in order to detect conflicts. Other log-based TM systems such as OneTM [2] and TokenTM [3] use different hardware to detect conflicts, but are based on the same mechanisms for logging. We believe that for the purposes of this work (studying version management alternatives), LogTM-SE is representative of all three systems so we will refer to it as log-based HTM from now on. The mechanisms

that our baseline log-based HTM implements in each of its dimensions are described in the following subsections.

3.1 Version Management

Log-based HTMs keep a copy of older values with their respective address in a software log, whereas transactional modifications update memory in-place. This software log must be initialized when a transaction begins. Each transactional store must follow three steps to ensure that the new value is in place and the old value is in the log: (1) the system brings the cache line to the processor if it is not already there, (2) the old data is stored in the log, and (3) the new data is stored in the cache.

The processor has specialized hardware to support efficient logging [21]. Even so, logging has a non-negligible cost, because it requires several movements to memory. In our system, we assume that a prefetch request with the log address is sent at the same time the transactional store is issued. This will bring the log entries close to the processor and it will accelerate the logging process.

Obviously, it is not necessary to restore those memory locations that have been modified by a committed transaction. On these occasions, the log is discarded. When a transaction has to abort, the recovery handler is called. The recovery handler is a software routine that walks the log in reverse order to restore the memory state by undoing all transactional memory updates.

3.2 Conflict Detection

The conflict detection mechanism permits the identification of those transactions that access the same memory locations. LogTM-SE uses the coherence protocol to detect conflicts and to ensure transactional isolation. Basically, the directory that tracks which lines are modified in private caches, forces signature checking when a remote request needs these lines. When a transactional line is evicted from a private cache, it is put on a sticky state in the directory, which assigns the ownership of the line to the evicting processor. If a sticky line is requested, the directory sends a message to the owner to force signature checking. This fact guarantees that all transactional requests will be checked even though the data has been evicted from the cache.

3.3 Conflict Resolution

The conflict resolution policy determines how a conflict is resolved, ensuring progress in the transactional execution. After detecting a conflict between two transactions, the conflict resolution policy stalls the requester, who waits until the other transaction commits. However, to avoid cyclical dependencies between stalled transactions that might produce deadlocks, transactions must inform a centralized cycle-detector when they are stalled. If a cycle occurs a transaction timestamp determines the younger transaction that participates in the cycle and aborts it. After the recovery mechanism, a backoff is performed to avoid multiple aborts of the same transactions.

4. ACCELERATED IMPLEMENTATIONS

One way to accelerate abort recovery is by providing specialized hardware to handle this situation, such as a fast hardware buffer to hold the log. Another alternative is to opt for a lazy version management solution, since in lazy version management systems old values stay in-place.

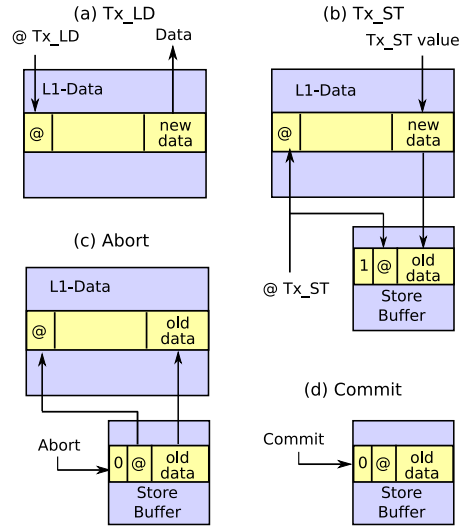

Figure 1: Tx Operations on Eager Implementation

We propose to use a gated store buffer, similar to Rock [12] and Crusoe^{TM} [9]. Our proposal allows both eager and lazy version management policies to be implemented on the same hardware, which allows the user to select the most appropriate execution mode according to the characteristics of the workload. For eager version management, the store buffer holds the old (pre-transaction) values, while for lazy version management it holds the new values (Crusoe ^{TM}and Rock use it this way). Stores kept in the buffer are "gated", until commit or abort time.

For a realistic design, we must implemented a solution that allows the execution of transactions of any size, *i.e.*, even transactions that overflow the buffer. Rock proposes to abort and re-execute overflowed transactions by software, whereas other unbounded lazy HTMs implement complex mechanisms to access overflowed transactional data.

We propose to use best-effort hardware for small transactions and an eager log-based mechanism for overflowing transactions in *both* eager and lazy version management implementations. This way, our system can take advantage of LogTM-VSE [18] techniques to survive page faults and context switches as well.

4.1 Eager Implementation

The eager implementation stores new values in place and the old values in the buffer. It operates identically to the log-based implementation, but it accelerates the version management mechanism with hardware support. Figure 1 shows how the store buffer is used to accelerate eager version management:

TxST: a transactional store sends the old data from the L1 to the buffer and simultaneously updates the L1 with the new data value (Figure 1b). At the same time, a CAM search is performed in the buffer using the store address. A match means that this address has been written before in this transaction and the correct data is already present in the buffer. If no match is found the old L1 data is stored in the first free entry of the buffer.

TxLD: transactional loads work identicaly to the original LogTM-SE proposal [21] (Figure 1a).

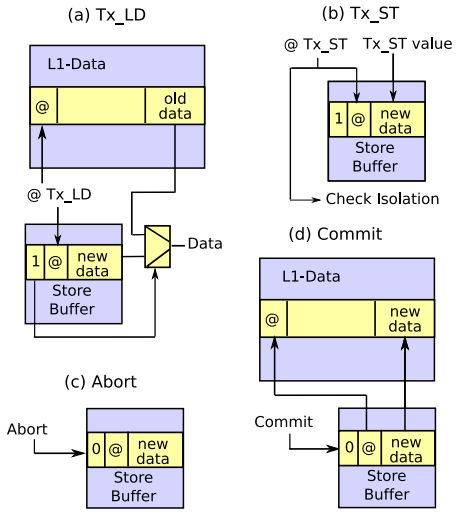

Figure 2: Tx Operations on Lazy Implementation

Abort: the processor is stopped and the state is restored by moving the old values from the buffer to the L1 using regular memory write requests (Figure 1c). If a line has been evicted from the L1 cache, it is brought from the lower levels of the hierarchy. This is not a problem, because the conflict detection mechanism guarantees that no other processor accesses the line (the abort process must be atomic). In our system, the cache has a single write port, so only one request can be sent at a time. When the abort recovery process finishes, the buffer is cleared.

Commit: all buffer entries are invalidated by flash-clearing the valid bits (Figure 1d). No other action is required.

Overflow: When the buffer overflows, transactions have to be recovered via the software log. In order to avoid unnecessary aborts, our eager implementation creates the software log *always*. Hence, transactional stores place the old values in both the store buffer and the software log. On overflow, a special flag is asserted and the store buffer is cleared. When a transaction aborts, the overflow flag decides if the transaction is recovered via hardware or software.

4.2 Lazy Implementation

The lazy implementation uses the hardware buffer to store speculative transactional values, keeping old values in the memory hierarchy. Hence, transactions do not modify the program state until they commit. This means that the L1 stores pre-transactional values, allowing an immediate recovery in case of abort.

This scheme is similar to RockTM, which uses the store buffer to store speculative writes [12]. Contrary to RockTM, our system is able to accelerate unbounded transactions using the existing hardware, by falling back to eager mode for transactions that overflow the buffer. Our approach implements the same conflict detection mechanism and conflict resolution policy as the eager implementation.

TxST: the buffer must hold the latest version of a memory location, while in a transaction. For this reason, a transactional store first performs a CAM search for the address in the buffer. If a match is found then the matching entry is updated with the new data, otherwise the data is stored in the first free entry of the buffer (Figure 2b). This operation is identical to that of a conventional store buffer in an out-of-order processor.

TxLD: similar to a load in an out-of-order processor, the transactional load does a CAM search with the address on the buffer in parallel to the L1 access (Figure 2a). If a match is found in the buffer, the data is forwarded from the buffer (independently of a L1 hit/miss).

Abort: all buffer entries are invalidated by flash-clearing the valid bits (Figure 2c). No other action is required. This is the same as the commit action of the eager implementation.

Commit: the processor is stopped and the state is commited by moving the new values from the buffer to the L1 using regular memory write requests (Figure 2d). If a line is not in the L1 cache, it is brought from the lower levels of the hierarchy. This is not a problem, because signatures are not released until the commit process finishes and the conflict detection mechanism stalls requesters of committing lines until the commit phase finishes. When the commit process finishes, the buffer is cleared. The commit operation of the lazy implementation is similar to the abort operation of the eager implementation.

Overflow: when a lazy mode transaction overflows the buffer the following things happen: first, the buffer is cleared (*i.e.*, we abort the transaction), then the buffer is put in eager mode, and last, the transaction is restarted (in eager mode). This means that an overflowed transaction executes using the eager implementation, as described in Section 4.1. Notice that aborts produced by overflows do not suffer the abort recovery penalty, because lazy version management recovers the state immediately. The only penalty is due to useful transactional work lost.

4.3 Discussion

Hybrid TM systems (and RockTM) recover overflowed transactions using software. This approach is useful if overflows are uncommon, but presents several drawbacks when overflows abound. First, hybrid TMs abort all the useful work done by an overflowed transaction. Second, overflowed transactions are executed entirely by software, which increases their execution time and generates contention. Last, HyTMs need mechanisms to guarantee isolation between software and hardware transactions, which may delay hardware transaction execution as well.

Contrary to hybrid TMs, our eager implementation does not discard useful work generated by overflowed transactions, and only aborted transactions suffer delays. In the lazy implementation, an overflowed transaction is only aborted once before re-executing in eager mode. There are cases though that these transactions may be aborted *before* they overflow the buffer, due to conflicts or other events. In such situations, we gain performance by using the buffer. Since our system does not use a software-only mode for overflowed transactions (a) it does not suffer the overheads of conflict detection among hardware and software transactions, and (b) overflowed transactions that commit without a conflict are accelerated.

Some lazy version management systems propose software buffering to handle overflows [1, 15, 17]. Software buffering is less effective for lazy version management than eager approaches: it would have to store the memory modifications of an overflowed transaction in a virtual memory structure that has to be accessed each time a memory location is loaded.

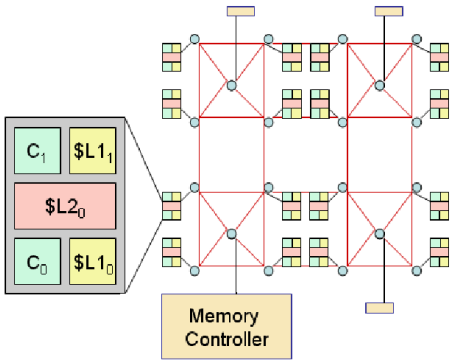

Figure 3: Base system configuration

Core	1.2 GHz in-order single issue
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, 4 memory controllers, 300-cycle latency
L2 directory	Bit vector of sharers, 6-cycle latency
Interconnect	Mesh, 64-byte links, 2-cycle wire latency, 1-cycle router latency
Signatures	2 Kb Parallel Chuckoo-Bloom filters

Table 1: Base system parameters

Recent proposals [15] accelerate this process using Bloom filters and look-up tables, but these techniques increment the complexity of the mechanism. On the other hand, eager log-based implementations offers an easy way to deal with overflowed transactions as we have seen.

5. RESULTS

For our experiments we assume a CMP processor with 32 cores. Each core is an in-order, single-issue SPARC processor with 4-way 32 KB private L1 instruction and data caches. The system has a mesh interconnection network that uses 64-byte links with adaptive routing. For our 32-core CMP design we have chosen to use a 16-node mesh, where each mesh node has two cores and a 1 MB shared L2 cache. This is a Non-Uniform Cache Access (NUCA) system, where a 16 MB L2 cache is distributed among the cores, as shown in Figure 3. Four memory controllers have been used to access the DRAM banks. 2 Kbit Read-Write and Write signatures have been used to track conflicts. Other system parameters are described in Table 1.

The base system and the different data version management mechanisms have been simulated using the Simics [11] simulation infrastructure from Virtutech and the Wisconsin GEMS toolset [13] to build the memory environment. GEMS provides a LogTM-SE implementation, which we have modified slightly to better model the latency of the stores to the software log.

In our evaluation of the realistic eager and lazy version management (labeled *Eager Implementation* and *Lazy Implementation*), we limit the store buffer to 32 entries. For the idealized eager and lazy version management (labeled *Ideal Eager* and *Ideal Lazy*) we assume an infinite store buffer.

Suite	Benchmark	Input parameters
ubench	Btree	10% insertions, 100K trans.
	Deque	5K dummy work, 100K trans.
SPLASH-2	Barnes	512 bodies
	Raytrace	teapot
STAMP	Bayes	16 vars, 512 records
	Genome	64K seg, 512 gene length, 32 seg. length
	Kmeans	15/15 clusters, 16K points
	Labyrinth	32*32*3 maze, 2048 routes
	Vacation	64K entries, 4K tasks, 16 items 60% queries, 90% user

Table 2: Benchmarks and input parameters

For our analysis we use applications from the SPLASH-2 benchmark suite [20], from the STAMP benchmark suite [5], and two microbenchmarks from the GEMS LogTM distribution. The applications and their respective input parameters are shown in Table 2.

The *ubench* benchmarks in Table 2 are the LogTM microbenchmarks. Both benchmarks perform basic operations on common data structures. They execute small transactions with variable contention.

SPLASH-2 is a suite of benchmarks for multiprocessors, where lock-protected regions have been transformed into transactional blocks [21]. As the SPLASH-2 benchmarks have been optimized over the years to avoid synchronization, most of the time is spent in small, fine-grained transactions.

STAMP [5] is the first benchmark suite written explicitly for transactional memory. The STAMP applications spend most of the execution time inside transactions. Transactions in STAMP are coarse-grain and operate on several data structures. STAMP is a benchmark suite in progress; we use version 0.9.7.

5.1 Log-based HTM Characterization

After studying transactional programs with different number of threads, we have observed that transactional behavior depends on the characteristics of the program. We classified transactional benchmarks in two groups in order to analyze applications with the same properties together.

Table 3 (top half of table) shows the properties of fine-grain benchmarks, where most of the time is spent in non-transactional code. The first column, labeled *Tx Time*, shows the time spent inside transactions as a percentage of the total execution time. The other three columns give us an idea of the average size of transactions. Column *Cycles/Tx* shows the average number of cycles per transactions, column *RdLn/Tx* shows the average number of cache lines read in a transaction, and column *WrLn/Tx* shows the average number of cache lines written in a transaction. Like fine-grain locking, only small sections, which access few lines, are executed in mutual exclusion. The behavior of these applications should be similar to transactional workloads written by expert programmers.

Table 3 (bottom half) also presents the characteristics of coarse-grain transactional applications, where all the time is spent in big atomic and isolated blocks. We believe that such applications are more typical of future transactional workloads. The numbers for Table 3 were collected by running the applications in single-thread LogTM-SE mode.

Figure 4, which presents the scalability of the log-based HTM with respect to a single-threaded LogTM-SE execu-

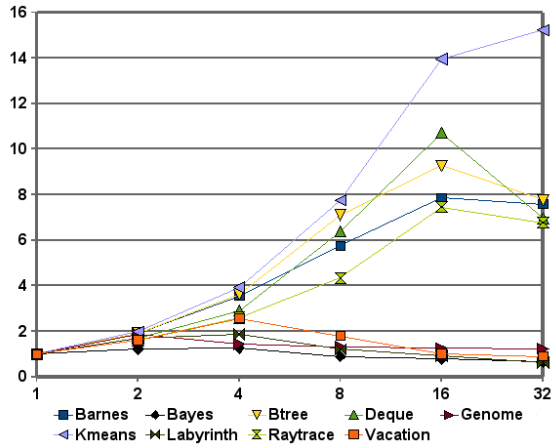

Figure 4: Log-based HTM speed-up

Benchmark	Tx Time	Cycles/Tx	RdLn/Tx	WrLn/Tx
Barnes	2.55%	713.59	6.29	4.61
Btree	56.84%	329.13	10.58	0.33
Deque	0.79%	62.05	2.50	2.89
Kmeans	8.75%	1481.5	7.62	2.75
Raytrace	0.15%	8.26	6.26	4.63
Bayes	85.25%	90179.17	65.96	46.13
Genome	98.71%	8428.51	37.35	9.77
Labyrinth	99.15%	17400.29	111.90	102.21
Vacation	88.98%	42543.35	179.31	22.88

Table 3: Fine-grain (top) and coarse-grain (bottom) benchmark properties

tion, shows that applications with small transactions, such as Barnes, Btree, Deque, Kmeans or Raytrace, have good performance when are executed with few threads. However, in 32-threaded executions the abort rate increases, preventing their scalability. Recent studies [19] have pointed out the shortcomings of log-based HTM when it has to deal with applications with huge transactions. Our characterization shows that benchmarks with these properties, like Bayes, Genome, Labyrinth or Vacation, present contention even when few threads are used.

For that reason, we evaluated these two categories separately. Moreover, we have decided to restrict our discussion to the configurations that make more sense for each category, that is for the 32-thread configuration for fine-grain applications and the 8-thread configuration for the coarse-grain applications. We chose 8-threads only for coarse-grain applications because, as we can see from Figure 4, they do not scale beyond this point.

Fine-grain applications scale better than coarse-grain applications because they execute non-transactional code most of the time. Figure 5, presents the normalized distribution of cycles in 32-threaded fine-grain programs (100% corresponds to total execution time). Log-based HTM corresponds to the left-most bar of each group.

On average, 41.2% of the execution time is spent outside transactions (labeled *Non Tx*), doing independent parallel work or waiting in barriers. However, useful transactional work is only 5.1% of the execution time (labeled *Good Tx*), which indicates high contention among transactions.

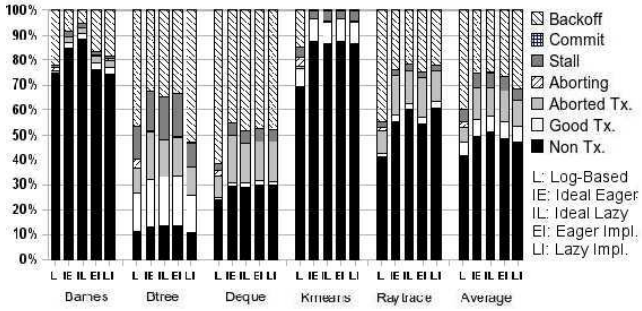

Figure 5: 32-thread fine-grain breakdown

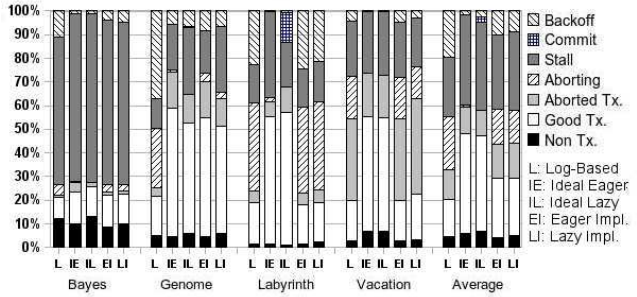

Figure 6: 8-thread coarse-grain breakdown

The biggest transactional overhead is the backoff, which consumes 40.2% of the execution time. This backoff is used to spread contention and guarantee the progress of the program, increasing exponentially with the abort rate. Although software recovery cycles only represents 2.5% of the execution time (labeled *Aborting*), the importance of backoff suggests high contention in some transactional regions.

Table 4 summarizes the abort rate of log-based and hardware HTMs, presenting the number of aborts per transaction in both 32-threaded fine-grain and 8-threaded coarse-grain executions. We can see how log-based HTM transactions require several executions before committing in the majority of the benchmarks, which prevents their scalability.

Contrary to fine-grain benchmarks, 12.5% of the time is wasted in aborted work (transactional cycles discarded when a transaction aborts) and 24.8% of the time transactions remain stalled after detecting a conflict (labeled *Aborted Tx* and *Stall* respectively). On average, 22.6% of the time is spent in abort recovery. This fact suggests that coarse-grain applications can increase their performance with version management techniques that reduce abort recovery time.

5.2 Acceleration Potential

Figures 7 and 8 show that the potential of using hardware version management policies is immense. Both eager and lazy ideal implementations improve log-based implementation in all the benchmarks because, as can be seen in Figures 5 and 6, they reduce the abort recovery time. This also reduces the abort rate (Table 4), because the number of conflicts that involve aborted transactions decreases.

Reducing the aborts has a positive impact in the performance of fine-grain applications. The ideal lazy implementation improves performance by 26.8% on average over the log-based implementation, reducing stall, backoff and aborted

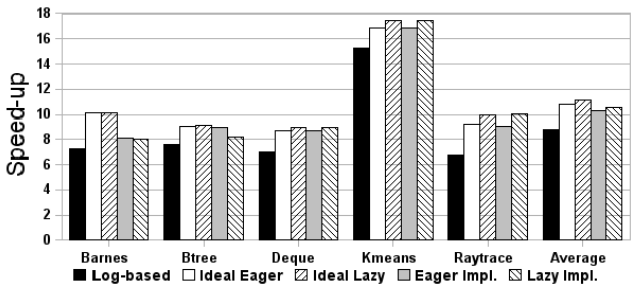

Figure 7: 32-thread fine-grain execution time

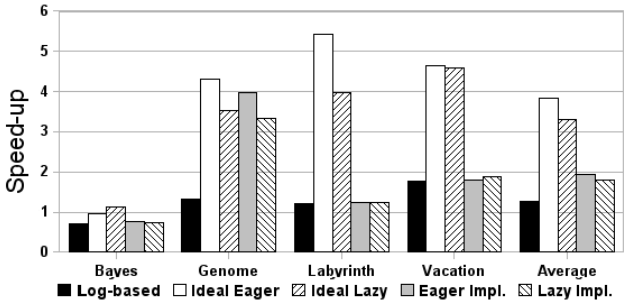

Figure 8: 8-thread coarse-grain execution time

cycles. Also, the ideal lazy implementation performs better than the ideal eager implementation in fine-grain benchmarks with high-contention because it offers a fast abort recovery mechanism (Figures 5 and 7).

Our experiments also show that hardware version management improves performance greatly over log-based HTMs in coarse-grain applications (Figure 8). These applications frequently abort large transactions, producing high contention. The slow abort recovery mechanism of log-based HTMs becomes the main bottleneck, delaying the execution of transactions and increasing the conflict rate (Figure 6).

For coarse-grain applications, the ideal eager implementation achieves a speed-up of 3.1X over the log-based implementation (Figure 8), encouraging hardware techniques for version management. In Figure 6, we can see that the ideal eager implementation only spends, on average, 0.8% of the execution time aborting. Contrary to fine-grain applications, for coarse-grain applications the ideal lazy implementation spends about 2.6% of the execution time in the commit phase. This is the reason why the ideal eager implementation improves performance by 15.9% over the ideal lazy implementation.

5.3 Accelerated Implementations

The small size of fine-grain transactions allows both the eager and lazy implementations with the 32-entry limited buffer to obtain similar performance to the ideal case. In Table 5 we show a breakdown of all the aborts in both fine-grain and coarse-grain applications.

For the eager implementation, the *SW* column shows the number of aborts recovered by the software handler, and the *Egr* column shows the number of aborts recovered by the store buffer. For the lazy implementation, *SW* are the aborts recovered by software, *Ovfl* are the aborts caused by an overflow of the store buffer, *Lazy* are the aborts recovered by the store buffer (no overflow), and *Egr* are the aborts recovered by the store buffer (after overflow).

Benchmark	Log-based	Eager Appr.	Lazy Appr.
Barnes	1.24	1.07	1.12
Btree	0.27	0.25	0.22
Deque	6.77	6.72	6.62
Kmeans	0.77	0.14	0.12
Raytrace	4.32	3.06	2.49
Bayes	4.08	2.03	2.12
Genome	3.48	3.68	3.61
Labyrinth	4.12	0.65	0.72
Vacation	3.06	0.53	0.56

Table 4: Aborts per transaction for 32-thread fine-grain (top) and 8-thread coarse-grain (bottom).

	Eager Impl.		Lazy Impl.			
Bench.	SW	Egr	Ovfl	Lazy	SW	Egr
Barnes	341	2035	402	1928	376	99
Btree	34	23261	2320	22156	38	2126
Deque	0	672387	0	635125	0	0
Kmeans	0	2692	0	2471	0	0
Raytrace	0	157903	0	117273	0	0
Bayes	710	1975	374	1393	677	102
Genome	1048	82166	369	84736	1191	4230
Labyrinth	954	599	2048	419	950	0
Vacation	12243	0	4091	0	9142	0

Table 5: Aborts breakdown for 32-thread fine-grain (top) and 8-thread coarse-grain (bottom).

Notice that the lazy implementation recovers instantaneously until the hardware overflows. On overflow, transactions are aborted and re-executed in eager mode. All the fine-grain applications, except Barnes and Btree, fit in the hardware buffer. Performance, compared to the log-based implementation, improves on average by 20.1%.

Although Table 5 shows that most of the coarse-grained transactions overflow the store buffer, our bounded lazy implementation can take advantage of the store buffer when re-executing in eager mode and a second abort is generated before the buffer overflowes again.

This happens in Barnes, Btree or Genome, where most of the aborts of overflowed transactions are recovered by hardware. Moreover, other applications with variable size transactions, like Labyrinth, can also be accelerated. However, some coarse-grain applications, such as Bayes or Vacation, almost always abort by software, which suggests the need for a bigger buffer. On average, the 32-entry buffer with eager version management improves coarse-grain application performance by 54.7% with respect to the log-based HTM, but its performance is still far from the ideal case.

6. CONCLUSIONS

In this paper, we have shown that aborts are frequent in many-threaded fine-grain and coarse-grain transactional applications and that this restricts severely the scalability of log-based HTMs.

To address this we have proposed the use of specialized hardware (specifically a gated store buffer) to accelerate version management. Furthermore, we have shown how to implement both eager and lazy version management using the same hardware (*i.e.*, the store buffer). For both, we have discussed how to utilize a log-based mechanism to handle buffer overflows. The use of a log-based HTM significantly simpli-

fies previous proposals that require complex structures to execute overflowed transactions while outperforming software-only recovery mechanisms.

Utilizing an infinite store buffer, we have shown that a lazy policy is better suited to applications with fine-grain transactions while an eager policy is better suited to applications with coarse-grain transactions. Utilizing a 32-entry store buffer, we obtain 20.1% average improvement over log-based HTM for applications with fine-grain transactions (using lazy version management) and 54.7% for applications with coarse-grain transactions (using eager version management).

7. 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*, Feb. 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*, Sept. 2008.

[6] 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.

[7] 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.

[8] 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.

[9] A. Klaiber. The Technology Behind Crusoe^{TM}Processors. Technical Report Technical report, Transmeta Corp., Jan. 2000.

[10] 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.

[11] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[12] K. M. Mark Moir and D. Nussbaum. The Adaptive Transactional Memory Test Platform: A Tool for Experimenting with Transactional Code for Rock. In *Procs. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[13] 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*, 33(4):92–99, 2005.

[14] 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.

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

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

[17] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *Procs. of the 1st ACM SIGPLAN Workshop on Transactional Computing*, June 2006.

[18] 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.

[19] 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.

[20] 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*, June 1995.

[21] 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.