Hybrid Transactional Memory to Accelerate Safe Lock-based Transactions

Enrique Vallejo *
University of Cantabria, Santander, Spain
enrique@atc.unican.es

Tim Harris
Microsoft Research, Cambridge, UK
tharris@microsoft.com

Adrián Cristal, Osman S. Unsal, Mateo Valero
Barcelona Supercomputing Center
{adrian.cristal, osman.unsal, mateo.valero}@bsc.es

Abstract

To reduce the overhead of Software Transactional Memory (STM) there are many recent proposals to build hybrid systems that use architectural support either to accelerate parts of a particular STM algorithm (Ha-TM), or to form a hybrid system allowing hardware-transactions and software-transactions to inter-operate in the same address space (Hy-TM).

In this paper we introduce a Hy-TM design based on multi-reader, single-writer locking when a transaction tries to commit. This approach is the first Hy-TM to combine three desirable features: (i) execution whether or not the architectural support is present, (ii) execution of a single common code path, whether a transaction is running in software or hardware, (iii) immunity, for correctly synchronized programs, from the "privatization" problem. Our architectural support can be any traditional HTM supporting bounded or unbounded-size transactions, along with an instruction to test whether or not the current thread is running inside a hardware transaction. With this we carefully design the Hy-TM so that portions of its work can be elided when running a transaction in hardware-mode.

While not compared with the native HTM system, our simulations show that, when running with HW support, the main runtime overheads of the STM system are elided: Depending on the workload, the speedup with read-only transactions is up to $3.03\times$ in the single-thread execution and $61\times$ in the 32-thread case, while with read-and-write transactions it reaches over $10\times$.

1. Introduction

Transactional Memory [12] aims to provide a simple programming interface to access shared data, avoiding some of the classical problems of shared memory concurrency. Hardware transactional memory (HTM) proposals (such as [2, 7, 22, 23]) can provide high performance and 'strong' semantics in which conflicts are detected between memory accesses made by transactions and memory accesses made directly. However, HTMs mainly rely on extending the coherence protocol with conflicts detected 'online' between the parties involved; this makes thread pre-emption and paging to disk complicated or impossible. Transactions that exceed local resources (cache capacity, write buffers) are either not supported, lead to complicated hardware designs, or highly increase the possibilities of high rates of false conflicts.

Software transactional memory (STM) proposals (such as [4, 6, 8, 9, 15]) allow the flexibility to explore different semantics and the possibility of deployment on current hardware. However, pure-software implementations suffer from high overheads. Even the simplest, blocking implementations of STM impose a significant slowdown. In addition to these performance overheads, STMs typically provide 'weak' semantics in which conflicts between transacted and non-transacted accesses go undetected. This leads to example programs like the 'privatization problem' that we discuss below; such programs are intuitively correctly synchronized but are not implemented with the semantics that programmers might anticipate.

The performance issues with STM have been examined in existing proposals for architectural support. One approach is hardware-accelerated TM (Ha-TM) in which an STM algorithm uses new hardware features to perform part of a transaction's work [21, 25, 27]. An alternative approach is hybrid TM (Hy-TM) [3, 11] in which the system supports the coexistence of HW and SW transactions, typically by starting a transaction in HW and re-executing it in SW if it overflows limited resources.

However, these previous approaches do not address the semantic problems that come from the 'weak' semantics offered by STM. The 'privatization problem' provides a motivating example:

```
//Thread 1                    //Thread 2
//x initially 0               atomic{ //Tx2
atomic { //Tx1                   if (x_shared) {
  x_shared = false;                 x = 42; //W2
}                                 }
x ++;  //W1                   }
```

In this example, x_shared is initially $true$, Thread 1 attempts to mark x as no longer shared (Tx1) before accessing it directly (W1), while Thread 2 attempts to check if x is still shared (Tx2) before accessing it inside its transaction (W2). This program is correctly synchronized under the 'single lock atomicity' model in which atomic sections are replaced by the use of a single process-wide lock. Programmers might therefore expect it to be correctly synchronized using TM and to see either $x = 1$ or $x = 43$. Unfortunately, as Spear *et al* discuss, existing STMs do not implement it correctly [28]. In some STM designs, Tx2 may be serialized before Tx1 but the write from W2 not yet be made back to main memory before W1 executes (allowing $x = 42$). In other STMs, Tx1 may conflict with Tx2 but Tx2 will continue running after the conflict is detected, letting W1 and W2 race (allowing $x = 0$ if Tx2 is rolled back after W1 executes).

Different systems have been proposed to overcome this and other similar problems that impact programming complexity. The work in [26] proposes a strongly-atomic java STM providing high performance by extensively using whole-program analyses and JIT optimizations to reduce the performance penalty of memory barriers in non-transactional code. In [16] the semantics of atomic blocks are defined by translating them into various lock acquire/re-

* While the author was at Microsoft Research.

lease implementations. The "single lock" translation is appealing because it provides semantics that are easy to explain in terms of existing programming language constructs and the existing Java memory model. However, the STM-based implementation of these single-lock semantics requires process-wide barrier operations at various stages during a transactions execution. This introduces contention between the implementations of transactions that access disjoint data.

Spear *et al* identify several techniques for avoiding the privatization problem: (1) Data can be statically partitioned between transacted and non-transacted parts of the heap, with explicit marshalling between them. (2) STM-aware libraries can be used for private accesses like W1 with the overhead that that entails. (3) Synchronization barriers can be added such that data is not used both in private mode and in shared mode between barriers. (4) Explicit fences can be required after transactions that make parts of the heap private. (5) Pessimistic concurrency control can be used so that if Tx1 is serialized after Tx2 then Tx1 will not commit until Tx2 has committed, or aborted and cleaned up. Of these techniques, only this last one has the desirable property that it does not impact the programming model. However, as Spear at al observe "PCC entails unacceptable overhead, primarily due to the overhead of reader locking".

In this paper we explore the role of architectural support in making a practical TM based on pessimistic concurrency control, thereby avoiding the privatization problem while also avoiding the need to develop a more complicated programming model involving barriers and fences that would not be needed under strong semantics. Our basic approach is a Hy-TM design in which we combine an optional, generic HTM system with a lock-based variant of Fraser's OSTM [6]. By structuring the HTM transactions carefully we allow them to interact correctly with software locks and for multiple transactions to hold the same read-mode lock concurrently.

Specifically, the main contributions of the paper are:

- A novel approach to handle multiple-reader, single-writer locks that allow HW transactions to elide them, while detecting conflicts with SW code acquiring them.
- A STM system based on the previous locks that can be accelerated with a bounded HTM, without the need of two different execution paths for HW and SW code.
- An evaluation of the different overhead sources of the base STM system, and an evaluation showing and quantifying the performance improvement obtained by HW-accelerating the different components of the system. Specifically, we quantify the overhead of the locking mechanism, the read-set management and the write-set management.
- An evaluation of a re-execution mechanism in HW transactions before changing to SW ones, showing that the optimal number of retries rises with the processor count.

The rest of the paper is organized as follows: section 2 discusses previous work in the area. Section 3 introduces the base lock-based STM system. Section 4 details how to leverage the locking mechanisms to allow for HW transactions that coexist with the previous SW ones, and how to avoid the main sources of overhead in the STM system. Section 5 shows our experimental setup and the performance results obtained. Section 6 discusses some specific points of the design, and finally Section 7 concludes the paper.

2. Related work. Hybrid TM proposals

Early STM implementations aimed for providing non-blocking progress guarantees. Ennals [5] argues that lock-based STM implementations are sensible given integration with the rest of the run-time system (to avoid, for example, a running thread spinning waiting for a lock to be released by a pre-empted thread). Lock-based implementations temporarily block a given resource, either from the encounter-point [9, 16] or in the commit phase [4] to simplify the design and provide higher performance. From these previous works, [9] and [4] can fail in problems such as the privatization one. The work of Menon *el al* [16] covers the problem, and proposes several alternative semantics for STM-only safe transactions. However, they rely on whole system and JIT optimizations, and barriers on volatile accesses. Our work is orthogonal to that one, being possible to adapt our HW acceleration proposal for their lock-based system.

Rajwar and Goodman [24] and Martínez and Torrellas [19] proposed mechanisms that allow threads to execute speculatively past locks without stalling for lock ownership. These systems support a traditional lock-based programming model rather than atomic transactions.

Hardware-accelerated STMs provide a performance speedup if the specific hardware they require is available. The work in [27] extends a traditional HW coherence protocol and cache operations to detect conflicts between transactions. In [25] new cache extensions are included so that transactions can avoid part of their bookkeeping work. Both proposals design specific HW support for a specific STM system.

Damron *et al's* hybrid system [3] is the first to consider HW acceleration by a traditional HTM system. However, their proposal compiles each transaction's code twice: once for HW-supported transactions and the second for SW-supported transactions. This system is based on hash-addressed ownership records (*orecs*), and they introduce the idea of extending HW transactions to detect conflicts with SW ones by checking the corresponding orec status. The improvement in PhTM [13] divides execution into different phases to allow HW-only execution. While we take a similar approach based on HW transactions observing the locks used by SW transactions, our proposal does not support such division. However, our pessimistic system introduces the idea of dynamically checking the HW transaction mode to require only one version of transacted code and we associate locks with objects to avoid the aliasing problems [31] caused by hashing. Another related hybrid system is NZTM [30], which does not require any indirection in HW-transactions, but is not addressed to lock-based transactions, as our proposal.

In [21] the authors propose a hybrid system based on TL2 [4]. It does not support software-only transactions. However, it does provide strong isolation, which is impractical to offer in a software-only scheme without whole-program analyses. As with our design, TL2 is based in a blocking, lock-based STM system. However, our design can still operate when hardware support is not available.

3. Overview of the base STM system

Our work is based on a lock-based variant of Fraser's OSTM [6]. This is an indirection-based STM in which transacted objects are represented by a header word that points to the object's current contents. Transactions run optimistically, building up thread-private logs of their tentative updates to objects. They commit by using the fair multi-reader single-writer variant of MCS spin-locks [20] to (i) lock the objects that they have accessed, before (ii) validating that there has been no conflicting update to any object read, and (iii) installing the tentative update to the objects being updated. Source-code to the base STM is available at http://www.cl.cam.ac.uk/research/srg/netos/lock-free/. The API, defined in Figure 1, is similar to other indirection-based STMs [10, 15]. All functions take a pointer to a per-thread state structure (`ptst_t`) from which per-thread memory management data structures are reached. Running transactions are represented by `stm_tx`

```
Begin transaction: stm_tx *new_stm_tx(ptst_t *ptst, stm *mem, sigjmp_buf *penv);
Commit Transaction: bool_t commit_stm_tx(ptst_t *ptst, stm_tx *t);
Validate transaction: bool_t validate_stm_tx(ptst_t *ptst, stm_tx *t);
Abort transaction: void abort_stm_tx(ptst_t *ptst, stm_tx *t);

Read STM block b: void *read_stm_blk(ptst_t *ptst, stm_tx *t, stm_blk *b);
Write STM block b: void *write_stm_blk(ptst_t *ptst, stm_tx *t, stm_blk *b);

Allocate STM block: stm_blk *new_stm_blk(ptst_t *ptst, stm *mem);
De-allocate STM block b: void free_stm_blk(ptst_t *ptst, stm *mem, stm_blk *b);
```

Figure 1. STM programmer interface

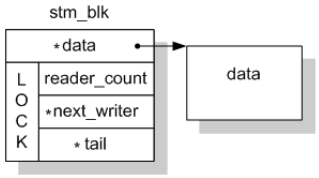


Figure 2. STM Block

transaction records. Transacted objects are represented by values of type `stm_blk`.

Transaction start. Transactions start with a call to `new_stm_tx()` that prepares the current transaction logs and records a return point (represented by a `sigjmp_buf`) to which to branch if the transaction becomes invalid.

Accessing objects in the transaction. The structure of a transactional object, `stm_blk`, is shown in Figure 2, where the locks behave as defined in [20] and the three lock fields have been dissected on the right. Each object must be "open" for read or write before using it. This open action searches the read and write log, and allocates a new entry if not found. Both read and write sets are implemented as ordered singly-linked lists of `stm_tx_entries` (depicted in Figure 3) using the `next` pointer. If the object is open for read, both `old` and `new` point to the current `data`; otherwise, a new `data` is allocated and `new` points to it. On each access, the pointer `new` is returned. Objects are allocated and de-allocated from per-thread free lists through `new_stm_blk` and `free_stm_blk`. As in Fraser's thesis, an epoch-based garbage collector is used to defer actual de-allocation until it is safe to do so [17].

Transaction commit. On commit, the STM acquires the locks corresponding to the objects in its log, using MRSW locks to allow different threads to commit concurrently if their read-sets overlap. If there are concurrent requests involving any writer, the locks use FIFO ordering on the queue. When all of the objects in the private log are locked, the system validates both the read and write sets. For reads, the validation merely consists of comparing the old pointers in the private log with the data pointers in the shared memory. Since the old pointer is copied in the first block access and only updated on a transaction commit, being equal to data means that no other

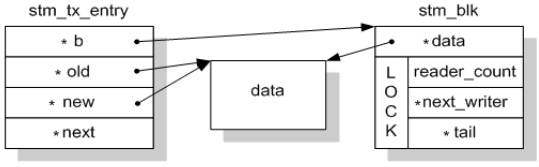


Figure 3. STM transaction entry

transaction has updated the block since the first access. In case of any conflict, the transaction releases all of the locks and aborts. On success, it commits the write set, updating the data pointers in shared memory to point to the private objects in the private log. Finally, all of the locks are released.

Transaction Abort. Ordinarily abort occurs when commit-time validation fails. In that case, locks are released and the transaction is re-executed. In addition, as with Fraser's other OSTM designs, a signal handler catches failures that may be generated by invalid transactions. The signal handler validates the transaction. If it is invalid, then the signal might have been generated by a race condition, so it is ignored and the transaction is aborted using siglongjmp, to the restore point created in the `new_stm_tx` call. As with other STM designs that allow transactions to run while invalid, this approach requires care from the programmer to ensure that invalid transactions will fail 'cleanly'. In managed languages like Java and C# all such failures can be detected in this way.

4. Acceleration opportunities with a generic HTM

The lock-based STM adds four main overheads when compared with running the same transactions on a native HTM (similarly to the analysis in [21]). First, the locking mechanism itself is not necessary in a HTM system. Second, transactions need to maintain the read-set and write-set lists. This introduces a list-search on each object accessed, and an increase in the memory used. In HTM systems the hardware itself tracks the objects accessed in the transaction (with read and write bits, signatures or other mechanisms). Third, on commit, the lists have to be traversed to lock and validate the objects. Fourth, the indirection-based object structure makes it necessary to copy entire objects when opening them for update, even if only a single field will be touched. In HTM these copies are managed implicitly and at a finer granularity.

We now consider how we can accelerate these costs by combining the STM with a generic HTM system. Our general approach is to modify the STM `new_stm_tx` and `commit_stm_tx` functions to start a 'sympathetic' hardware transaction when each transaction is started. The transaction is then initially attempted in hardware mode. It invokes the same STM-library operations as normal, allowing us to avoid compiling the transaction's implementation twice. However, the library operations recognise opportunities to remove operations that are redundant while the HW transaction is active. If the HW resources are exceeded then the HW transaction is aborted and we can fall back to SW execution.

In Section 4.1 we clarify the assumptions that we make of the HTM and then, in Sections 4.2-4.4 we present three possible acceleration mechanisms.

4.1 Requirements on the HTM system

We use an ordinary HTM supporting strong atomicity [1] between transacted and non-transacted accesses. We assume that all memory accesses are implicitly transacted when running inside a transaction. In addition we assume that the ISA provides an `InHWTx` mechanism to determine whether or not execution is inside a transaction. When making performance-related decisions we assume that per-cache conflict detection is used and that updates are made in-place (as with LogTM). These assumptions affect performance, not correctness.

4.2 First acceleration: avoiding locking

Our first observation is that locking is often un-needed when running a transaction in HW. When a collision with other transaction happens:

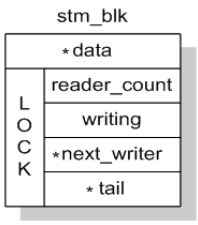


Figure 4. Modified STM_blk with writing field in the lock

- With other HW transaction, the underlying HTM detects the conflict on the conflict location, and forces one of the HW transactions to abort or wait. Thus, the use of locking is unnecesary. Furthermore, lock acquire/release operations can introduce false conflicts between pairs of HW transactions accessing disjoint parts of the same objects.
- With a SW transaction holding a given lock when the HW transaction tries to acquire it (supposing the original behaiour), the lock mechanism in the HW transaction would spin-wait until the lock becomes free. However, HW transactions can never spin: when the SW transaction releases the lock, the update of the spin location would conflict with the HW transaction read set, constituting an isolation violation, and forcing a HW abort.

However, we cannot simply remove all locking: SW-locked objects must be respected by HW transactions, and SW transactions must not acquire a lock if this conflicts with a HW transaction accessing the object. Our approach is to modify the lock and unlock operations so that HW transactions test for collisions with SW (a SW writer in the case of a HW `read_lock` operation, and any reader or writer in a HW `write_lock` operation). To this end, we modify the lock structure adding a new "`writing`" field (as depicted in Figure 4) which is set by SW writers when they acquire the lock, and cleared on release. The rest of the algorithm remains as in [20]. With this mechanism, the correct actions required by HW and SW lock operations are defined in Table 1.

SW reader	original behaviour in [20], with increase of `reader_count`
SW writer	original behaviour in [20] and set `writing=true` after lock acquisition
HW reader	check `writing==false` and exit (no field updated); otherwise, HW abort.
HW writer	check `writing==false` and `reader_count==0`, and exit; otherwise, HW abort

Table 1. Locking operations

This ensures that HW transactions do not interfere with locked elements, while eliding most of the locking work (and allowing multiple HW transactions to "acquire" the same lock without conflict in read mode). Considering a direct-update, early detection HTM system like LogTM [22](which we use in our evaluations), the coherence extension that provide the strong atomicity of the HTM will prevent SW transactions from acquiring a lock in a conflicting mode once it has been checked by a HW transaction. Table 2 shows the actions involved when the lock is held by a thread in reader or writer mode, and a new writer arrives, which generates a collision (the case of a thread holding the lock in writer mode and a reader arriving is analogous). Table 3 shows the case of two readers, in which the system allows both threads to continue. In a system with lazy detection (like TCC [7] or Bulk [2]) the actions would be different, but the behavior would be still correct.

To enable the different behaviour depending on the HW or SW mode, we define a function `in_HW_Tx()`, that, using the `inHWTx`

(a) Writer - writer

	Lock holder:	HW writer	SW writer
	Lock status:	*Reader_count = 0 Writing = false*	*Reader_count = 0 Writing = true*
HW writer	Check:	*Writing & reader_count*	*Writing & reader_count*
	Action:	HTM triggers abort on real data collisions	Abort after check of *Writing.*
SW writer	Action:	Coherence extensions prevent SW from modifying *Writing*	Use of the ordinary queue system

(b) Writer - reader

	Lock holder:	HW reader	SW reader
	Lock status:	*Reader_count = 0 Writing = false*	*Reader_count = 1 Writing = false*
HW writer	Check:	*Writing & reader_count*	*Writing & reader_count*
	Action:	HTM triggers abort on real data collisions	Abort after check of *Reader_count.*
SW writer	Action:	Coherence extensions prevent SW from modifying *Writing*	Use of the ordinary queue system

Table 2. Lock acquiring with one or two writers involved, HW and SW cases.

ISA instruction, returns true if the transaction is executed with HW support, or false if it is SW-only. With it, the code for read and write lock and unlock operations is presented in Figure 5 (omitted original parts are those in [20]):

```
Read_lock (lock, qnode) {
  if (in_HW_Tx()) {
    if (!lock->writing)
      return; //Succeed
    else ABORT_HW_TX();
  }
  else { [..] //Original code }
}
```
a)

```
Write_lock (lock, qnode) {
  if (in_HW_Tx()) {
    if (!lock->writing)&&
    (lock->read_count ==0)
      return; //Succeed
    else ABORT_HW_TX();
  }
  else { [..] //Original code }
}
```
b)

```
Read_unlock (lock, qnode) {
  if (in_HW_Tx()) { ;}
  else { [..] //Original code }
}

Write_unlock (lock, qnode) {
  if (in_HW_Tx()) { ;}
  else { [..] //Original code }
}
```
c)

Figure 5. Modified a) Read_lock, b) Write_lock and c) unlock operations.

Of course, we must do this carefully to avoid false sharing within the lock data structures. The problem is illustrated by considering two readers: a HW Tx checking `lock.writting` and a SW Tx trying to update `lock.reader_count`. If both fields are in the same cache line, the collision detection from the HTM system will consider them to conflict. In this case it effectively prevents HW and SW readers from acquiring the same lock in concurrently in read mode. To prevent this, we have analyzed the possible sources of false sharing in the code and padded structures where necessary (such as the state structures `psts_t` or the `stm_blk` headers).

	Current lock holder:	HW reader	SW reader
Arrives	Lock status:	*Reader_count = 0 Writing = false*	*Reader_count = 1 Writing = false*
HW reader	Check:	*Writing*	*Writing*
	Action:	No real conflict. HTM does not trigger any abort	Proceed after check of *Writing.*
SW reader	Action:	Coherence extensions allow SW to modify *Reader_count*	Use of the ordinary queue system, *reader_count = 2*

Table 3. Lock acquiring with two readers involved, HW and SW cases.

4.3 Second acceleration: do not maintain a read set

Our second acceleration technique comes from the fact that the explicit read-set list can be elided in HW transactions. HW transactions still need to check the locks of read objects for two reasons: to avoid reading write-locked objects and to prevent any further SW Tx write-locking the object. Therefore, in HW transactions, instead of building up a read-set list the `read_stm_blk` checks that the lock is in the appropriate status (`writing = false`) and returns the `data` pointer to the shared object. This prevents any further software writer committing changes to the block before the HW commit, because of strong atomicity.

This operation reduces the overhead for two reasons:

- The transaction log is reduced to the write set. Though a search on each access is still needed, it is much faster.
- The same applies to the validation: the number of validation steps is reduced to the modified objects count only.

4.4 Third acceleration: make updates in place

The idea in the previous subsection, when applied to the write set, would prevent the object copy on access and provide updates in place. However, in this case there is a limitation: as specified in section 3, SW transactions rely on the update of the `data` pointer to detect conflicts on the validation step. Directly accessing objects during HW transactions without updating the data pointer (because no new version of the object would be allocated) wouldn't let SW transaction detect HW updates.

To overcome this, we use the same idea as in the previous section plus an additional `version` counter in the object header. This field is set to 0 when the object is first instantiated (in any HW or SW transaction) and increased on every call to `write_stm_blk` by a HW transaction. Software transactions copy the value of this word in the entry on the read and write sets and the validation process implies checking both the `data` pointer and the `version` field. With this optimization, HW transactions make their updates in place, do not maintain read or write sets, and consequently avoid any searching through these sets.

Special care is needed to handle `version` overflows. A simple solution is to abort HW transactions on counter overflow, and clear the counter on SW updates (to further increase the time between overflows).

5. Evaluation

The proposed system has been implemented and simulated with GEMS [18], a simulator based in the full-system simulator Simics [14], making use of the LogTM MESI protocol. Network parameters have been set to resemble those of the Sunfire E25K [29]. The simulator was modified to restart a transaction in SW mode after a fixed number of HW attempts The number of threads on each simulation always equals the number of processors.

We extended the STM implementation to use the LogTM ISA, modifying only the STM library and not the test harness calling it. The library was also modified so that HW transactions never call the memory allocation and free functions in the GC commented in Section 3; instead, the HW transaction is aborted if a local pool of pre-allocated chunks is exhausted. There are three reasons to do this: It prevents some limitations on the base simulator, although the HTM simulates unbounded transactions there are various low level processor operations (such as TLB misses) which cannot occur transactionally; this action prevents congestion problems in the GC, and eventually, the OS, when a HW transaction modifies some global structure; finally, it models a HTM system more restricted than the original, unbounded, LogTM model.

We simulated four versions of our design:

- The original, software only, STM system. (`sw`).
- The version eliding the locks, but maintaining both read and write sets (`rw`).
- The version that avoids entries in the read set (`noread`).
- The version without read or write sets (`nowrite`).

5.1 Evaluation benchmarks

We used the two existing microbenchmarks from Fraser's thesis work: a red-black tree and a skip-list, with sizes specified by a key k, similarly to other works in the area [4, 6]. Each thread executes read transactions that search and read a given key, or write transactions that add (update) or remove the given key, with a certain probability p. The update and remove operations may need to rebalance the tree or correct the links in the skip list, introducing some contention between operations on different keys. In every case the root of the tree or list is always part of the read set of the transactions. After a sufficient warm-up period, we measure the number of simulated cycles per transaction, averaged across nine simulation runs and executed for a period long enough to converge to a fixed value. The performance will be the inverse of this value, and in most cases, our results are shown normalized against the single-processor, SW-only result.

5.2 Performance results

Our first test shows the performance improvement with a single-processor, reflecting the sequential work removed by the HW-support. Table 4 shows these values for different data structure sizes, using workloads with read-only transactions ($p = 0\%$) and for $p = 10\%$.

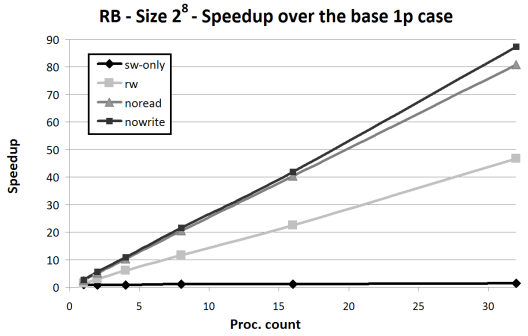
	RB p = 0			RB p = 10%			Skip p = 0			Skip p = 10%		
Key k	8	11	15	8	11	15	8	11	15	8	11	15
sw	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
rw	1.45	1.43	1.41	1.42	1.43	1.41	1.32	1.33	1.32	1.31	1.33	1.31
noread	2.53	2.74	2.76	2.38	2.68	2.69	2.10	2.47	2.75	2.04	2.41	2.70
nowrite	2.70	2.93	2.90	2.67	2.90	2.86	2.30	2.73	3.03	2.31	2.69	3.02

Table 4. Single processor performance

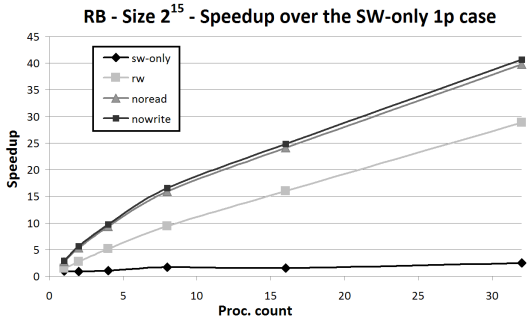
The elided steps constitute a substantial part of the execution time, given that up to $3.03\times$ speedup is obtained only from them being removed. We also observe that with larger problem sizes, the improvement is higher than with small sizes: the memory usage is also lower (no need for read or write sets) so the number of cache misses, which increases with the problem size, is reduced.

Figure 6 shows the multi-processor performance obtained with read-only transactions in the RB tree for key sizes 8 and 15. As in the table, the figures are normalized to the performance of the single-processor, base STM system. Note that, considering those initial single-processor values for each curve, there is no super-lineal speedup in the plots in Figure 6(a), though the overall speedup exceeds the processor count.

The small workload shows a linear speedup in all the HW-accelerated cases. The baseline offers low performance and poor scalability, but the lock elision (`rw`) removes this, leading to a performance increase of $33.04\times$ (in the 32p case). The `noread` line shows that avoiding the read-set entries (and the associated search) leads to a $57.19\times$ speedup from the base system. Finally, avoiding the read and write set copies gives further improvements, largely through the simplifications made to the commit processing; even in the case of read-only transactions there is an overall $61.81\times$ speedup (in the 32p case). The graph below is similar, but shows how the cache limits reached for the larger workload reduce the scalability of the system presented. The low scalability of the base system can be explained by profiling the `sw` and `nowrite` cases in Figure 7(a).

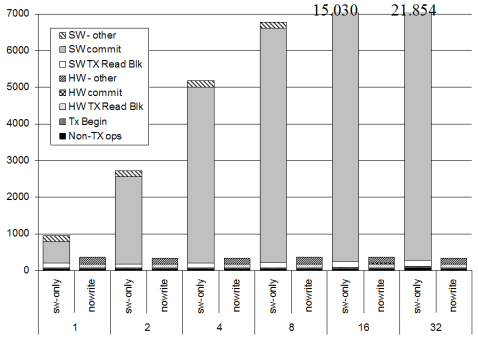


(a) Key 8

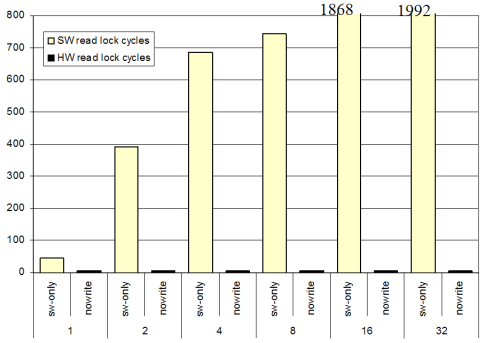


(b) Key 15

Figure 6. Performance of the RB with key sizes 8 and 15.



(a) Cycle dissection



(b) Lock cycles

Figure 7. Read-only transactions cycle dissection and cycles spent locking and unlocking.

The first point to note is that the commit phase from the sw design is almost removed in the `nowrite` approach: the validation and update steps are unnecessary. Also, this commit phase in `sw` grows with the number of processors: this comes from the cycles spent manipulating read-set locks. As seen in Figure 7(b)), this time grows, as expected, with the number of processors due to the queue operations on the locks and contention on the `reader_count` field - especially toward the root of the shared structure. In the case of `nowrite`, the lock action is reduced to a simple check.

Figure 8 shows the performance obtained with a skip-list under low (a, $k = 15$ and $p = 10\%$) and high contention (b, $k = 8$ and $p = 10\%$). The large workload with low contention suffers a small performance decrease from the read-only case (similar to Figure 6(b) but it still scales well, reaching $10.81\times$ speedup in the 32p case against the base SW case. The performance from `noread` to `nowrite` is quite different in this case, where the write set is not empty. With high contention (Figure 8(b)) the performance degrades as the number of processors increases. This degradation comes from the higher proportion of slower, SW transactions, due to the HW transactions abort rate increasing (as shown in Figure 9(a)). Though the actual HW abort rate is not very high, the fact that LogTM makes processors wait rather than abort on conflicts makes these figures quite significant for the overall performance. The graph shows how the lines tend to the base sw-line as the abort rate increases.

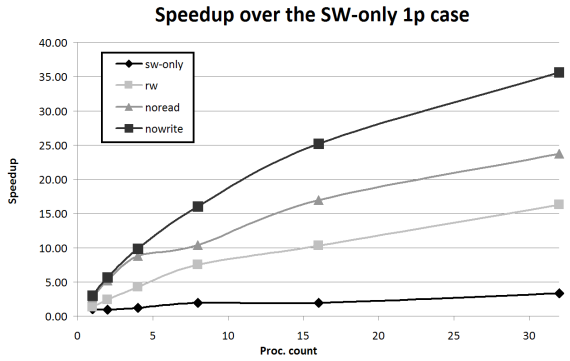
An improvement for this case might be to re-execute transactions in HW mode some number of times (as suggested in [3]). We evaluated the performance of this, varying the maximum re-execution-count, simulating the `nowrite` case from Figure 8(b). Figure 10 shows the performance results for different retry counts, normalized to the case of no retrying (`No retries`, which corresponds with `nowrite` in Figure 8(b)). As seen, the performance is generally improved in the congested area, up to a maximum of 33% in the 32p case. However, the optimal value for retries is low (2 to 4, darker bars) with a low number of processors (4 to 8), and increases (5 to 10, lighter bars) with a higher processor count (16 to 32). We observed a similar behaviour in `rw` and `noread`. This suggests that a fixed re-execution policy is not ideal. This result is understood by considering that, with high thread counts, the collisions are coming from different HW transactions, while in other cases they come from different sources, such as collisions with SW transactions, or resource exhaustion. We might therefore consider further architectural extensions to reflect the reasons for HW transactions aborting.

6. Discussion

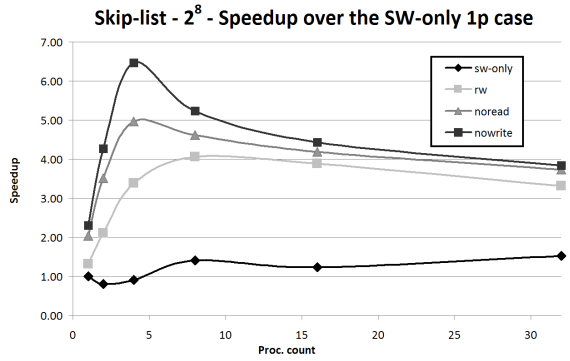
In this section we discuss some points of our design:

- *Use of MRSW queue-based locks in the STM*: The use of these locks in an STM is unusual: they introduce the need of updating a field (`lock.reader_count`) even for read blocks on software transactions, and this might generate a performance loss due to contention in the lock variable. Figure 11 shows the performance impact of these locks: we simulated the skip list-2^8 with a read-only workload and the locks removed (`no_locks`). Performance results are better than the HW-accelerated version that merely avoids the locks (in this case there are no locks at all) but worse than our optimized `nowrite` proposal, that is not limited to read-only transactions. The high contention in the acquisition and releasing of the read locks (especially close to the tree root) generates a significant overhead, as previously presented in Figure 7(b).

  The use of MRSW locking is motivated by the programming model: they allow us to implement the examples like the 'privatization problem' from the introduction while retaining a simple

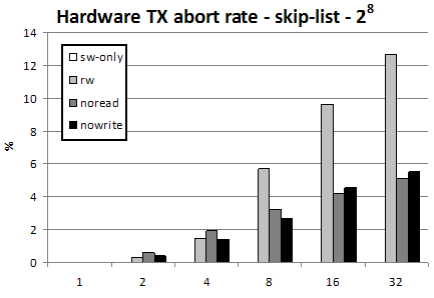


(a) Low contention

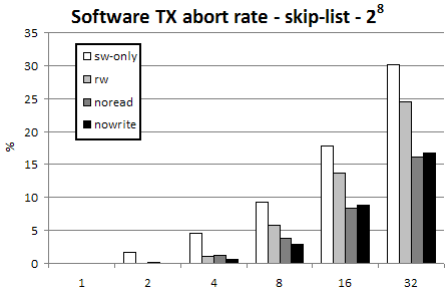


(b) High contention

Figure 8. Performance under low and high contention, $p = 10\%$.



(a) HW aborts



(b) SW aborts

Figure 9. Transactions aborted in hardware and software modes, $p = 10\%$.

programming model and the ability to operate on existing machines without hardware support. Although the worst-case cost of locking is worse than other STM designs (such as Fraser's lock-free variant of OSTM), the presence of the HTM acceleration introduced in this paper allows short-running transactions to execute in hardware mode (without needing to acquire these locks), while long-running transactions spend proportionately more of their time executing (rather than committing) and so the locking costs incurred are less significant.

- *Starvation of software transactions*: An open question is whether SW transactions might suffer from starvation when they have constant collisions with HW transactions. If the HTM system resembles LogTM (as in our simulation) this might happen, given that the coherence extensions would make a SW transaction stop while there is a collision with any running HW transaction. In our experiments, software transactions in the 32p highly contended case presented in Figure 8(b) would run on average 400 to 500 times slower than HW-accelerated ones, mainly due to this temporary starvation. On the other hand, in a system with lazy detection (such as TCC [7]), the result would be the opposite: any SW update of a memory location generating a collision would abort remote HW transactions. Eventually, this might be considered as an issue in HTM systems, but our assumption of a generic HTM system prevents us from considering it in this work.

7. Conclusion

This paper has introduced a new lock management system, so that locks used by the STM implementation can be easily avoided in atomic blocks, in which they are unnecessary. Based on this lock mechanism, we have presented a Software Transactional Memory system that can be accelerated in presence of any traditional HTM system, but still works safely without it.

Evaluations show that the base system is limited by the semi-visibility of readers, but the HW-acceleration removes this limitation, and provided optimizations remove the main overheads of the STM system. On our benchmarks, we have seen that the optimal number of HW retries after an abort is not very high, and increases with the number of processors. This point might be interesting for future HTM systems, to implement variable number of retries on their control interface, and then "fall back to software", to simplify their design.

Finally, the congestion control between HW and SW transactions has appeared as an important point of the design. While our proposal always gives priority to HW transactions (in a LogTM-like environment), it seems interesting that the system can control the priorities of different threads in the system to prevent starva-

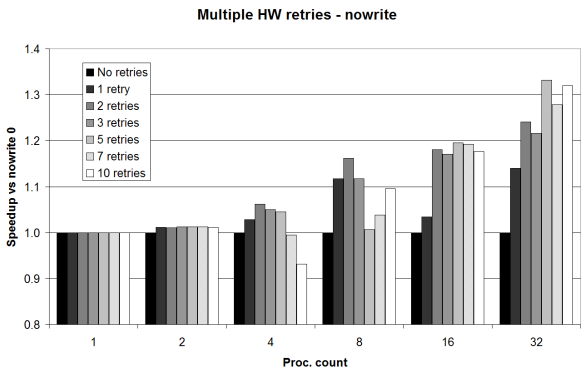


Figure 10. Mutiple HW retries of aborted transactions

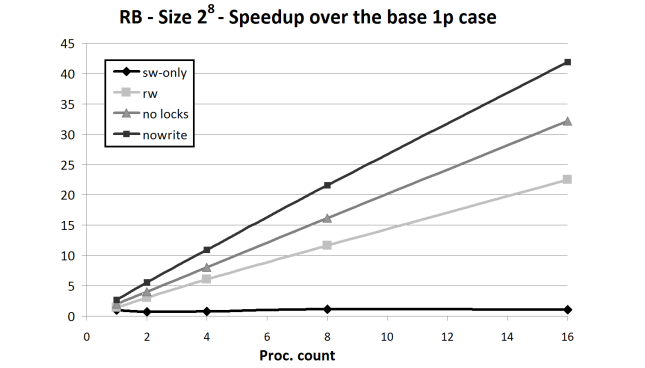


Figure 11. Study of the locking overhead - read only

tion, for example by imposing some restriction on HW-accelerated transactions.

Acknowledgments

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2004-07440-C02-01, TIN2007-60625 and TIN2007-6802-C02-01, and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). The authors would like to thank the valuable comments received from Satnam Singh and the anonymous reviewers.

References

[1] Colin Blundell, E Christopher Lewis and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), July 2006.

[2] Luis Ceze, James Tuck, Calin Cascaval and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In the Proceedings of the 33rd *Intl. Symposium on Computer Architecture (ISCA)*, June 2006.

[3] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir and Daniel Nussbaum. Hybrid transactional memory. 12th *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In the *Proceedings of the 20th Intl. Symposium on Distributed Computing (DISC)*, Stockholm, Sweden, Sept. 2006.

[5] Robert Ennals Software Transactional Memory Should Not Be Obstruction-Free. Intel Research Cambridge Technical Report. IRC-TR-06-052.

[6] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, Vol 25, Issue 2, May 2007

[7] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis and Kunle Olukotun. Transactional Memory Coherence and Consistency. In the Proceedings of the 31st *Intl. Symposium on Computer Architecture (ISCA)*, Munich, Germany, June 2004.

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In the 18th *Conf. on Object-oriented Programming, Systems, Languages, and Apps. (OOPSLA)*, Anaheim, CA, 2003.

[9] T. Harris, Mark Plesko, Avraham Shinnar and David Tarditi. Optimizing Memory Transactions. In the Proceedings of the *Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[10] M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In the 20th *Intl. Symposium on Computer Architecture (ISCA)*, May 1993.

[11] Sanjeev Kumar, Michael Chu, Christopher J. Hughes and Partha Kundu. and Anthony Nguyen Hybrid Transactional Memory. In the 11th *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[12] J. Larus and R. Rajwar. Transactional Memory. Morgan Claypool Synthesis Series, 2007.

[13] Yossi Lev, Mark Moir and Dan Nussbaum. PhTM: Phased Transactional Memory. *Workshop on Transactional Computing (TRANSACT)*, 2007.

[14] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50-58, February 2002.

[15] V. J. Marathe, M. F. Spear, A. Acharya, D. Eisenstat, W. N. S. Iii and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In the Proceedings of the 1st *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Canada, June 2006.

[16] Vijay Menon *et al*. Towards a Lock-based Semantics for Java STM. *University of Washington Technical Report*: UW-CSE-07-11-01. November 2007.

[17] M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 15, No. 6, pp 491- 504, June 2004.

[18] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN)*, Sept. 2005

[19] J.F. Martinez and J. Torrellas, Speculative synchronization: applying thread-level speculation to explicitly parallel applications, Proc. 10th *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, Oct. 2002.

[20] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. Proceedings of the 3rd *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1991, Williamsburg, Virginia, United States

[21] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In 34th *International Symposium on Computer Architecture*. San Diego, June 2007.

[22] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill and David A. Wood. LogTM: Log-Based Transactional Memory. In the Proceedings of the 12th *Intl. Conference on High-Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2006.

[23] R. Rajwar, Maurice Herlihy and Konrad Lai. Virtualizing Transactional Memory. In the Proceedings of the 32nd *International. Symposium on Computer Architecture (ISCA)*, Madison, WI, June 2005.

[24] R. Rajwar and J. R. Goodman. Speculative Lock Elision: enabling highly concurrent multithreaded execution. In Proc. Of the 34th *Intl. Symposium on Microarchitecture*, Austin, Texas, 2001.

[25] Bratin Saha, Ali-Reza Adl-Tabatabai and Quinn Jacobson. Architectural Support for Software Transactional Memory. In the Proceedings of the 39th *Intl. Symposium on Microarchitecture (MICRO)*, Orlando, FL, Dec. 2006.

[26] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steve Balensiefer, Dan Grossman, Richard Hudson, Katherine F. Moore and Bratin Saha. Enforcing isolation and ordering in STM. In the Proceedings of the 2007 *ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*. San Diego, California, 2007.

[27] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra Marathe, Sandhya Dwarkadas and Michael L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. Proc. of the 34th *International Symposium on Computer Architecture (ISCA)*, June 2007.

[28] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro and Michael L. Scott. Privatization Techniques for Software Transactional Memory. UR CSD;TR915. Rochester University, Feb. 2007.

[29] Sun Microsystems. Sun Fire E25K/E20K Systems Overview. Technical Report 817-4136-12, 2005.

[30] Fuad Tabba and Cong Wang and James R. Goodman and Mark Moir. NZTM: Nonblocking, Zero-Indirection Transactional Memory. *Workshop on Transactional Computing (TRANSACT)*, 2007.

[31] C. Zilles and R. Rajwar. Transactional Memory and the Birthday Paradox. In the 19th annual *ACM symposium on Parallel algorithms and architectures (SPAA)*. June 2007.