Hybrid Transactional Memory with Pessimistic Concurrency Control[1]

Enrique Vallejo[1], Sutirtha Sanyal[2], Tim Harris[3], Fernando Vallejo[1], Ramón Beivide[1], Osman Unsal[2], Adrián Cristal[2] and Mateo Valero[2]

[1]*University of Cantabria, Avda. Castros S/N, Santander, Spain*
{enrique.vallejo, fernando.vallejo, ramon.beivide}@.unican.es
[2]*Barcelona Supercomputing Center, C/Jordi Girona, 31, 08034,Barcelona, Spain*
{sutirtha.sanyal, osman. unsal, adrian.cristal, mateo.valero}@bsc.es
[3]*Microsoft Research, J J Thomson Avenue, Cambridge, UK.*
tharris@microsoft.com

Abstract

Transactional Memory (TM) intends to simplify the design and implementation of the shared-memory data structures used in parallel software. Many Software TM systems are based on writer-locks to protect the data being modified. Such implementations can suffer from the "privatization" problem, in which transactional and non-transactional accesses to the same location can lead to inconsistent results. One solution is the use of Pessimistic Concurrency Control, but it entails an important performance penalty due to the need of reader-writer locking.

In this paper a hybrid TM design is proposed to reduce the performance overheads caused by the use of these locks while combining three desirable features: (i) full TM functionality whether or not the architectural support is present; (ii) execution of a single common code path in software or hardware; and, (iii) immunity from the privatization problem. The analysis shows how a Hybrid TM can lose important properties, such as starvation freedom. To overcome this issue, Directory Reservations is presented, a low-cost mechanism improving existent solutions designed for Hardware TM.

Keywords: Hybrid Transactional Memory; Pessimistic Concurrency Control; Writer Starvation; Directory Reservation

[1] © Springer, 2011. This is the author's version of the work. The final publication is available at Springer via http://dx.doi.org/10.1007/s10766-010-0158-x

1. Introduction

Transactional Memory (TM), [19], 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 [3], [13], [29], [30]) can provide high performance and 'strong' semantics in which conflicts between memory accesses made by transactions and memory accesses made in non-transaction mode, are detected. 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 notably increase the possibilities of high rates of false conflicts.
Software Transactional Memory (STM) proposals (such as [5], [10], [14], [15], [22]) 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. One approach to improve performance is hardware-accelerated TM (Ha-TM) in which a STM uses new hardware features to perform part of the transaction's work [28], [32], [35]. An alternative approach is hybrid TM (Hy-TM) [4], [18], 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.
In addition, STMs typically provide 'weak' semantics in which conflicts between transacted and non-transacted accesses go undetected. This leads to programs that, intuitively, are correctly synchronized but they are not implemented with the semantics that programmers might anticipate. One example is the 'privatization problem' as presented below:

```
// Thread 1
// x initially 0
atomic{ // Tx1
        x_shared = false ;
}
x++; // W1
```

```
// Thread 2
atomic {                    // Tx2
        if ( x_shared ){
    x = 42 ; // W2
        }
}
```

In this example, with *x_shared* 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, existing STMs do not correctly implement it as Spear et al discuss in [36]. In some STM designs, Tx2 may be serialized before Tx1 but the write from W2 might 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 [34] 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 [23] the semantics of atomic blocks are defined by translating them into various lock acquire/release 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 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.
Several techniques are identified in [36] 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 in the previous example, with the overhead that it entails. (3) Synchronization barriers can be added such that data is not used both in private and shared modes between barriers. (4) Explicit fences can be required after transactions that make parts of the heap private. (5) Pessimistic Concurrency Control (PCC) 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 affect the

programming model. However, as observed in [36], "PCC entails unacceptable overhead, primarily due to the overhead of reader locking".

The current paper is devoted to explore the role of architectural support in making a practical TM based on PCC. The proposed system avoids the privatization problem while eliminating 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 [10]. By carefully structuring the HTM transactions, we allow them both to correctly interact with software locks and to hold the same read-mode lock concurrently by multiple transactions.

Specifically, the main contributions of the paper are:

- An evaluation of the base PCC STM system, showing the impact of the different overheads. The resulting system is compared with a non-blocking STM that provides higher performance at the cost of failing in the privatization problem.
- A novel approach to handle multiple-reader, single-writer locks that allows HW transactions to elide them and detects conflicts with SW code.
- A Hybrid TM system based on the previous lock handling that can be accelerated with a generic bounded HTM, without the need of two different execution paths for HW and SW transactions. The evaluation quantifies the overhead of the locking mechanism, the read-set management and the write-set management. It also shows that the original fairness, provided by queue-based locks, is lost when using a HTM.
- Directory Reservations is proposed, which is a low-cost hardware mechanism to regain fairness between reader and writer transactions, both hardware and software, preventing starvation and blocking.

3. Overview of the base STM

Our work is built over a lock-based variant of Fraser's OSTM [10]. 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 (MRSW) variant of MCS spin-locks [27] to (i) lock the objects that they have accessed, before (ii) validating that there has been no conflict, and (iii) updating objects modified in the transaction. Source-code of 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 [17], [22]. Running transactions are represented by tx_id 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 branch to if the transaction becomes invalid.

Accessing objects in the transaction. The structure of a transactional object, stm_blk, is shown in Figure 3. The locks behave as defined in [27]. The three lock fields have been dissected on the right side. 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 2) 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 [24].

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. When all of the locks are taken, the system validates both the read and write sets. This merely consists of comparing the old pointers in the private log with the data pointers in the shared memory, since the

```
Begin transaction: stm_tx *new_stm_tx(tx_id *tx, stm *mem, sigjmp_buf *penv);
Commit Transaction: bool commit_stm_tx(tx_id *tx);
Validate transaction: bool validate_stm_tx(tx_id *tx);
Abort transaction: void abort_stm_tx(tx_id *tx);

Read STM block b: void *read_stm_blk(tx_id *tx, stm blk *b);
Write STM block b: void *write_stm_blk(tx_id *tx, stm blk *b);

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

Figure 1: STM programmer interface

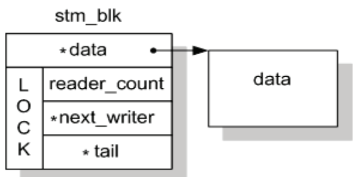


Figure 3: stm_blk structure

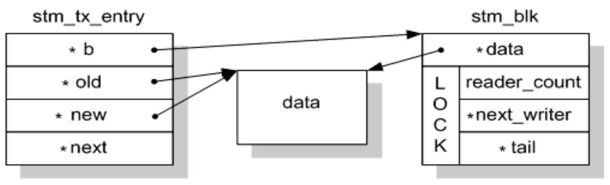


Figure 3: stm_tx_entry structure

old pointer is copied in the first block access and only updated on a transaction commit. 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 newly allocated data blocks. 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. As with 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 returning to the restore point created in the new_stm_tx call using siglongjmp. As with other 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 C# or Java all such failures could be detected by the runtime system.

4. Hybrid-TM

A lock-based STM adds four main overheads when compared with running the same transactions on a native HTM (as in [28]). 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 for each object accessed, and an increase in the used memory. 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 is going to be touched. In HTM these copies are managed implicitly and at a finer granularity.

Next, it is proposed the construction of a hybrid TM system that removes these costs by combining the base STM with a generic HTM system. A progressive approach is used: initially, the STM new_stm_tx and commit_stm_tx functions are modified to start a ‘sympathetic’ HTM transaction when each transaction is started. Transactions are initially attempted in this ‘hardware mode’, being called hardware-transaction or HW-Tx. The HW-Tx will follow the same execution path as the original code: It invokes the same STM-library operations as normal, preventing double compilation of the transaction’s implementation. If the transaction aborts a given number of times, it will be retried in the original, slow SW-only mode without the wrapped HTM transaction. Both HW and SW transactions will be allowed to run concurrently in the system.

Many runtime operations are unneeded when running in the HW-accelerated mode. Instead of providing two different execution paths (for HW-Txs and SW-Txs) we will progressively analyze the different steps that can be dynamically removed from the HW-accelerated execution path. In any case, if HW resources are exceeded then the HW transaction is aborted and fallen back to SW execution. This progressive approach allows us to observe the implications of every architectural change and its effect on the global system performance. In addition, we will compare the system with a faster ordinary STM which fails in the privatization problem.

We use an ordinary HTM supporting strong atomicity between transacted and non-transacted accesses [2]. We assume that all memory accesses are implicitly transacted when running inside a transaction. We also assume that the ISA provides a new instruction InHWTx to determine whether or not execution is inside a HW 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 latest assumptions affect performance, not correctness. Next, Sections 4.1 to 4.3 present three progressive accelerations for the HW-based execution path.

4.1 First acceleration: avoiding locking

As the underlying HTM mechanisms provide transaction atomicity and data collision detection, locking is often un-needed when running a transaction in HW. However, we cannot simply remove all locking: SW transactions require locks for correctness, and SW-locked objects must be respected by HW-Txs. Also, SW-Txs should not acquire a lock if this conflicts with a HW-Tx accessing the object.

Our approach is based on maintaining the lock access but modifying the lock and unlock operations (and the lock structure). In HW mode, transactions just test for collisions with SW ones: a HW read_lock operation will conflict with a previous SW writer, and a HW write_lock operation will conflict with any SW reader or writer. 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 for SW-Txs remains as in [27]. This prevents HW-Txs from accessing any object being transactionally modified by a SW-Tx: if a conflict is detected, the HW-Tx aborts. The detailed correct actions required by HW and SW lock operations are defined in Table 1.

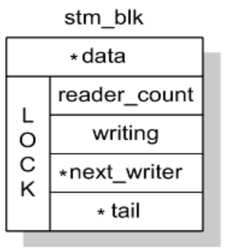


Figure 4: Modified stm_blk with writing field

SW reader	original behaviour in [27], with increase of reader_count
SW writer	original behaviour in [27], 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

In a direct-update early detection HTM system (such as LogTM [29] used in our evaluations), the coherence extensions providing strong atomicity also prevent any SW-Tx from acquiring a lock in a conflicting mode. Table 2 shows the actions involved when the lock is held by a thread in reader or writer mode and a new writer arrives generating a collision (the case of a thread holding the lock in writer mode and a reader arriving is analogous). When two transactions try to lock the same object in read mode, they can proceed in parallel: SW-Txs just increase the reader_count field, while HW-Txs check the writing field. Using a HTM with lazy detection (like TCC [13] or Bulk [3]) the specific actions would be different but the behavior would be still correct.

	Current Lock holder:	HW writer	SW writer

	Lock status:	reader_count=0
		writing=false
			reader_count=0
			writing=false

HW writer	Check:	writing== false & reader_count ==0	Writing==false & reader_count==0
	Action:	HTM aborts on real data colisions	Explicit abort after check of writing
SW Writer	Action:	HTM coherence extensions prevent SW-Tx from modifying writing	Use of the ordinary lock queue system

a) Writer-writer conflict

	Current Lock holder:	HW reader	SW reader

	Lock status:	reader_count=0
		writing=false
			reader_count=1 (+)
			writing=false

HW writer	Check:	writing== false & reader_count ==0	writing== false & reader_count ==0
	Action:	HTM aborts on real data colisions	Explicit abort after check of reader_count
SW Writer	Action:	HTM coherence extensions prevent SW-Tx from modifying writing	Use of the ordinary lock queue system

b) Writer-reader conflict

Table 2: Lock acquisition with writers involved

To dynamically modify the lock behaviour, we define a function in_HW_Tx(), that, using the inHWTx ISA instruction, returns true only if the code is being executed with HW support. According to it, the code for read and write lock and unlock operations is the one presented in Figure 5 (omitted original parts can be found in [27]):

```
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

4.2 Second acceleration: skip the read set

Our second acceleration technique comes from the fact that the explicit read-set list can be elided in HW transactions. Such 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, instead of building up a read-set list, in HW-TXs the read_stm_blk call will check that the lock is in the appropriate status (writing = false) and return 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 decreases 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.

4.3 Third acceleration: in-place updates

Applying the idea of removing the private log to the write set would prevent the object copy on the first access and would provide updates in place. However, as specified in Section 3, SW transactions rely on the update of the data pointer to detect conflicts on the validation step. HW transactions do not need to allocate a new data block, but modifying the object without updating the data pointer wouldn't allow SW transactions detect HW updates.
To overcome this drawback, we add 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-Tx. SW-Txs copy the value of this word in the entry on the read and write sets. The validation process implies checking both the data pointer and the version field. Special care is needed to handle version overflows. A simple solution is to abort HW-Txs on counter overflow and clear the counter on SW updates (to minimize overflows).
With this last optimization, HW transactions make their updates in place, do not maintain read or write sets, and consequently avoid any search on these sets. As our experimental results will show later, this last system provides the highest performance.

5. HyTM Evaluation

The proposed system has been implemented and simulated with GEMS [25], a tool based in the full-system simulator Simics [21], using a MESI LogTM protocol. Network parameters have been set to resemble those of the Sunfire E25K [37]. Each HW-Tx was retried for 3 times before switching to SW-only. The modelled system contains 16 processors and the thread count was variable, while making sure to always leave an empty processor for OS tasks.
We extended the STM implementation to use the LogTM ISA, modifying only the STM library and not the applications using it. The STM library was modified so that HW-Txs never call the system malloc and free functions in the garbage collector. Instead, HW transactions are aborted if a local pool of pre-allocated chunks is exhausted. Three reasons support this. First, it prevents some limitations on the base simulator. Although the HTM simulates unbounded transactions, there

are various low level operations (related with OS locking) which cannot occur transactionally. Second, it prevents congestion problems in the GC, and eventually, in 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), which is expected to be limited by the cost of read-locking.
- A version eliding the locks in HW, but maintaining both read and write sets (rw).
- A version that avoids entries in the read set (noread).
- A full accelerated version, without read and write sets (nowrite),

A real HyTM system would only use the last version, but we simulate all of them to evaluate the effect that each change introduces. We also profiled Fraser's STM [10], which provides a higher performance than the SW-only lock-based STM, at the cost of failing in the privatization problem. Its comparison points out the performance costs of the privatization safety property based on pessimistic concurrency control.

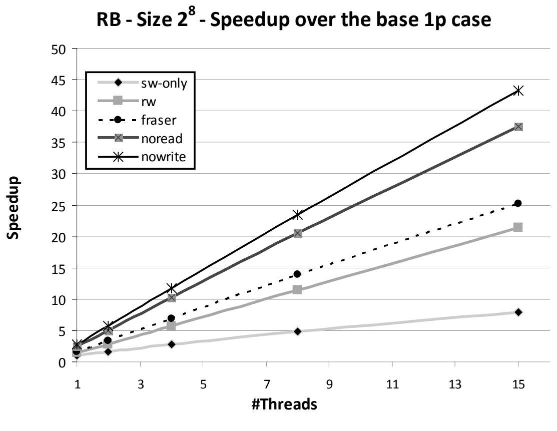
5.1 Evaluation benchmarks

We used three microbenchmarks: red-black (RB) tree, skip-list and hash-table. The problem sizes are specified by a key *k*, similarly to other works in the area [5], [10]. Each thread executes write transactions with probability *p,* or read transactions that search and read a given key with probability (1-*p*). In the first two benchmarks, write transactions will add or remove a given key. Write operations may need to rebalance the tree or to correct the links in the skip list, introducing some contention between operations on different keys. In these programs, the root of the tree or list is always part of the transactions read set. In the hash-table microbenchmark, we implemented two instances of linked hash tables with different size. Each writing transaction will move the given entry from one table to the other (if found) or add/remove the key (after searching on both tables, if needed) with the same probability. In this case, there is no reader-congestion in the root of the table (actually, there is no such root).

In all cases, after a sufficient warm-up period, the number of simulated cycles per transaction was measured, averaged across nine simulation runs and executed for a period long enough to converge to a fixed value. Performance is reported as the inverse of this value. In most cases, our results are normalized against the single-processor, SW-only performance.

5.2 Performance results

Our first test shows the performance improvement obtained by a single processor, which reflects the sequential work removed by the HW-support. Table 3 shows these values for different problem sizes (from 2^8 to 2^{15} maximum elements), using workloads with read-only transactions, p = 0%, and for p = 10%.
Fraser's performance improves the base system in a 20-60% as there is no reader-lock overhead. The lock elision (sw) provides a similar but slightly lower improvement, given that the writing lock field still has to be checked by readers. The remaining two improvements provide higher benefits (up to 3.45) by offloading the log management and conflict detection functions to the HTM. This improvement grows with the problem size (2^{11}, 2^{15}) in RB and Skip, as the transaction log size is also increased.
Figure 6 shows the multi-processor performance obtained with read-only transactions in the RB tree and hash table for key size 8. The skip list performance is similar to the RB tree, and different problem sizes show similar behaviour, so those plots are omitted. Plots are normalized to the performance of the single-processor, base STM system. Considering the values in Table 3 there is no super-linear speedup. Both workloads show a constant speedup, but the baseline STM offers much lower scalability in the RB benchmark. In the hash-table benchmark,



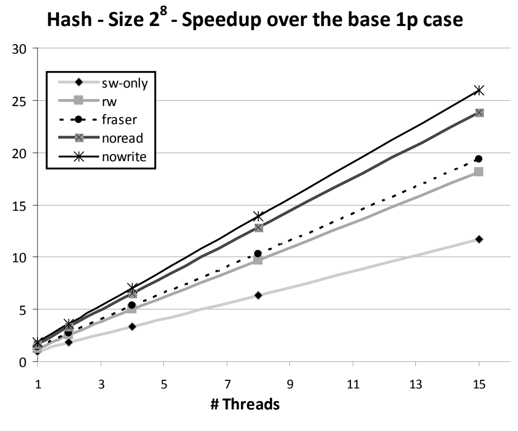


Figure 6: Speedup of read-only RB (left) and Hash (right) with key size 8

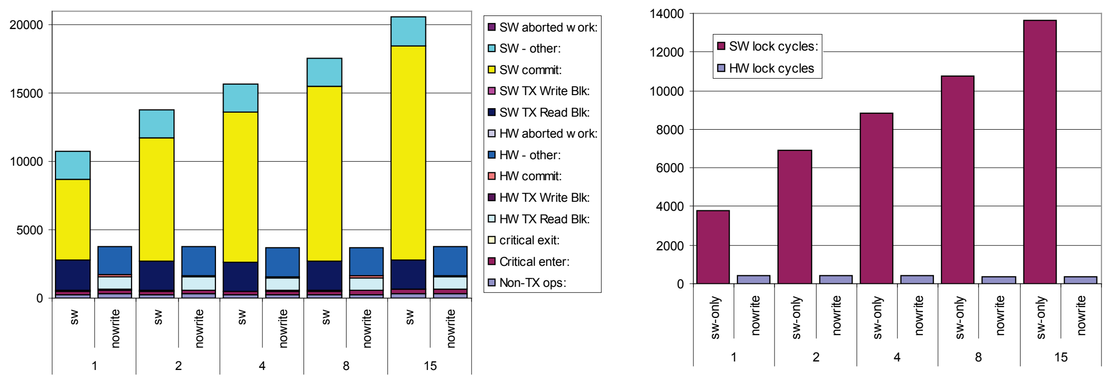


Figure 7: RB read-only transactions cycle dissection (left) and cycles spent with lock operations

for each thread count the performance improvement of the HyTM models over the base STM is roughly constant, with the values presented in Table 3. In the RB benchmark the performance improvement rises with the thread count: with 15 threads lock elision (rw), gains a factor of 2.72x, while noread (that saves the read set storage and validation) and nowrite (that saves any logging and commit steps) provide speedups of 4.77x and 5.49x.

The poor scalability of the base STM system in the RB benchmarks can be explained by profiling the sw and nowrite cases in Figure 7. The left plot dissects the execution steps, showing how the commit phase of the sw design is almost removed in the nowrite approach as the validation and update steps are unnecessary. Also, this commit phase grows in sw with the number of processors due to the cycles spent manipulating read-set locks. To verify this, the right plot shows how lock handling time grows with the thread count due to the contention on the lock reader_count field, which happens especially toward the root of the shared structure. In the case of nowrite, the lock action is reduced to a simple check that does not introduce coherence contention and remains constant with the thread count.

	RB *p*=0			RB *p*=10%			Skip *p* = 0			Skip *p* = 10%			Hash *p* = 0		Hash *p* = 10%	
Key *k*	8	11	15	8	11	15	8	11	15	8	11	15	8	11	8	11
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	1.00	1.00	1.00	1.00
rw	1.43	1.41	1.38	1.41	1.38	1.33	1.03	1.08	1.13	1.04	1.05	1.17	1.23	1.23	1.23	1.33
fraser	1.70	1.69	1.65	2.19	1.23	1.22	1.14	1.23	1.25	1.15	1.18	1.26	1.38	1.39	1.03	1.49
noread	2.50	2.78	2.92	2.22	2.27	2.35	1.93	2.24	2.37	1.94	2.12	2.58	1.71	1.71	1.52	1.75
nowrite	2.84	3.20	3.36	1.92	2.53	2.68	2.29	2.67	2.84	2.83	3.16	3.45	1.88	1.87	1.68	2.00

Table 3: Single processor performance

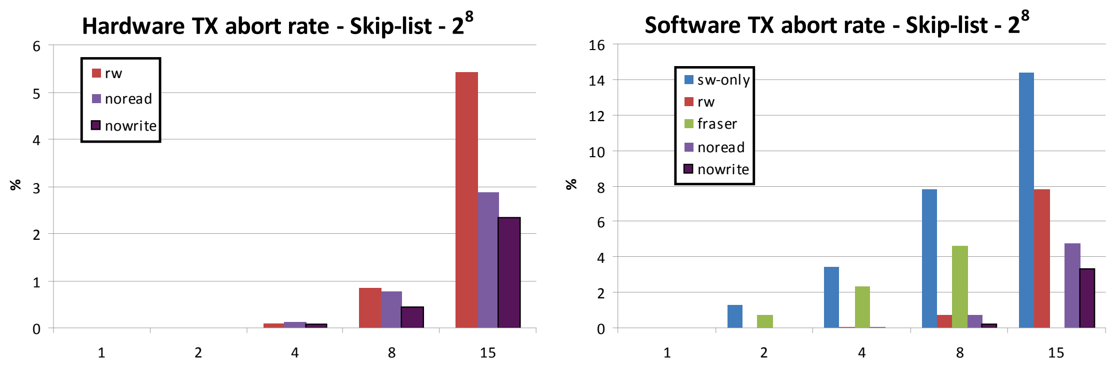


Figure 9: Transactions aborted in hardware and software modes, p = 0%.

Figure 8 shows the performance obtained with the skip-list benchmark under low (left, *k=15* and *p = 10%*) and high (right, *k =8* and *p = 10%*) contention. With low contention, Fraser's model scales well, but we observe how the hybrid versions' performance degrades with the number of threads. With higher contention (right), the performance of all models suffers as expected, but the performance penalty is much higher in the hybrid models. This degradation comes from two facts. First, it is due to the higher proportion of slower SW transactions triggered by the increase of the rate of HW transactions aborted, as shown in Figure 9. Though the actual HW abort rate is not very high, LogTM makes processors wait on conflicts rather than abort, leading to a significant impact on the overall performance. The second fact is the lack of fairness that will be studied in the next Section.

Figure 10 shows the performance in a contended case of the hash-table benchmark ($k = 8$, $p = 25\%$). The base STM performs better than before, as there is no root-node reader congestion. The right plots show that, as collisions are more infrequent in this data structure, the abort rate is much lower and hence, the scalability higher. The system maintains, basically, the single-processor

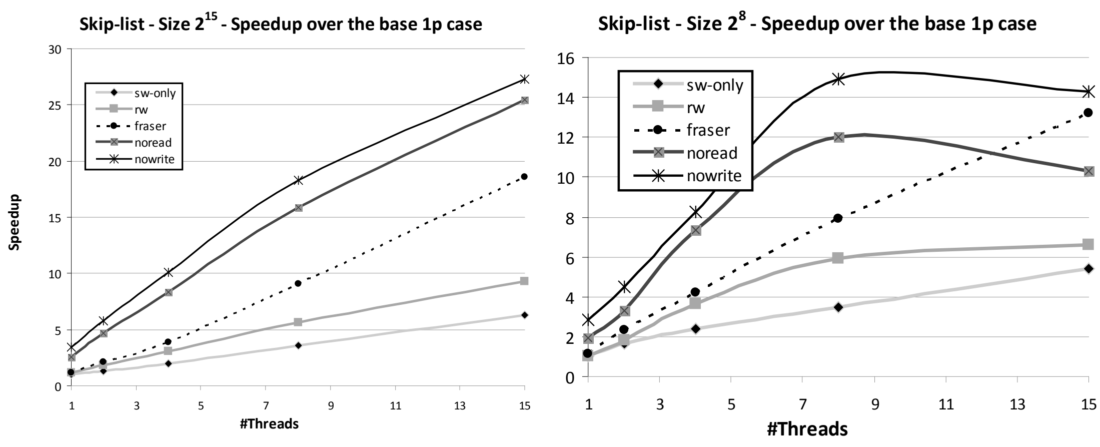


Figure 8: Skip-list performance under low (left) and high (right) contention, p = 10%

performance speedups for every thread count.

5.3 Fairness issues

The base STM provides fairness guarantees between different reading and writing transactions. The use of per-object queue-based FIFO locks at commit time implies that a transaction that wants to modify an object will have to wait for any previous committing readers (or writers) to release the lock. In addition, any subsequent transaction that reads the same object will have to wait for the writer to finish, later on the queue.

However, we found that on the HyTM system this fairness guaranty is lost when using LogTM as the base HTM (or, in general, with any eager conflict detection HTM that stalls the requestor of conflicting addresses). The problem comes from the way that LogTM extends the coherence mechanism to provide atomicity: Whenever a coherence request arrives at a given node, if such request conflicts with the ongoing transaction, a NACK ("Negative Acknowledgement") reply is sent to the coherence requestor, temporarily denying access to the line. This prevents the requester from reading or writing transactionally modified lines.

This NACKing mechanism effectively provides a hardware-based lock on those lines read or modified during the transaction. It can be either read-locking when a transaction reads a line, in which case several transactions can concurrently access the shared line, or write-locking if the line has been modified by a transaction and is kept with exclusive coherence permissions.

This kind of multiple-reader, single-writer locking can lead to writer starvation on frequently read lines. The problem can occur if two processors are continually running transactions that hold the same cache line in their read set while a third processor is waiting to make a transactional write to that line. The putative writer will be continually NACKed while the readers continue executing transactions. A

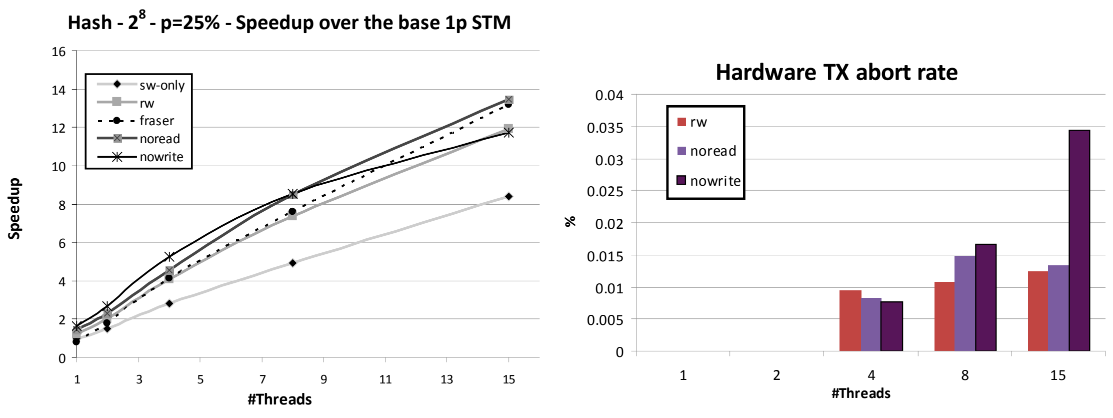


Figure 10: Hash-table speedup and abort rate with k=8, p=25% (high contention)

}

Figure 11: Example of code that stalls due to writer starvation

pathological example is shown in the code of Figure 11, where *a* is a shared variable initially set to 0, *N* is the thread count and *th_id<N* is a per-thread id. When there are enough threads running this code, execution never ends due to writer starvation. Without special contention management, even four threads are enough to block the Hybrid TM system.
Similarly, we analyzed the performance of individual processors in the red-black tree microbenchmark. With 32 threads we found that, after starting a couple of thousand transactions, and depending on the program run, from 4 to 12 processors are stalled trying to modify some node which is frequently read because of its location close to the root. This starvation anomaly does not happen in the base STM. In the next Section, we introduce the idea of Directory Reservations, a novel, low cost HW mechanism that builds a “semi-fair” queue to prevent this type of problems.

6. Directory Reservations

In Figure 12, a request (Step 1) from *C* is forwarded (2) to A and B transactional readers, and NACKed (3). The general idea of Directory Reservations is that such NACKed requests will issue a request to the directory to reserve the line, piggybacked in the coherence response. The directory is extended with a new R (Reserved) bit per line, and an attached Reservation Table (RT) to support the new functionality, as presented in Figure 13. Figure 12 also shows the new behaviour: After receiving a reservation request (4) from processor *C*, if a line is available in the RT the directory sets the R flag for the line, allocates such empty line in the RT and records the address and requestor processor id *C* in the requestor field. The fields read_count and W will be discussed later.
Whenever any other processor *D* issues a GETX (“get exclusive”) or GETS (“get

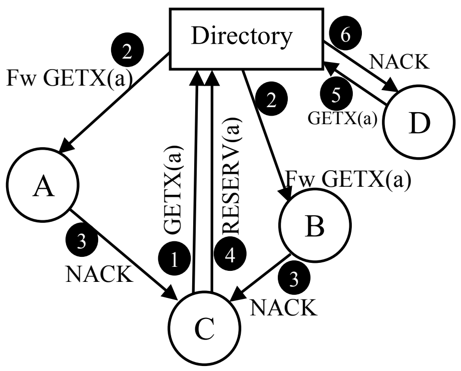


Figure 12: Reservation Mechanism

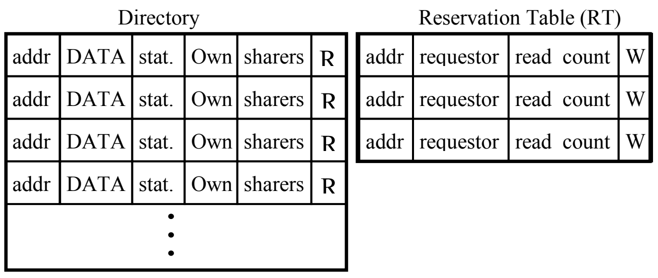


Figure 13: Reservation Table structure

shared") request for the same line (step 6 in Figure 12), it will arrive at the directory controller where the R flag is already set. The directory controller will get the corresponding line from the RT and compare the requestor id, *D*, with the saved requestor field containing *C*. Being different, the controller determines that the current requestor is not the one that reserved the line, and sends a NACK message (step 7) to *D* without any need to forward the request to the current sharers. Only requests from processor *C* (the one that reserved the line) will be forwarded to the corresponding processors (owner or sharers). If *C* receives new NACK replies, it will have to repeat the request until it is successfully satisfied. Eventually, in a general case, the blocking processors (A and B) will commit their transaction. When this happens, no new NACK will be issued to the coherence requestor, so *C* will receive the valid data with the valid permissions. In this point, the final message from *C* to the directory clears both the *R* flag and the current requestor, and finishes the reservation.

In practice, we must be careful because the processors executing *A* or *B* may themselves incur a conflict with a transaction executing in *D* (conflict in another line different from *a*). This would cause a deadlock, e.g.: *A* waiting for *D*, *D* waiting for *C*, and *C* waiting for *A*. This can be addressed by extending the existing deadlock avoidance mechanism used in the HTM. In the specific case of LogTM (deadlock detection based on timestamps), it can be accomplished by never NACKing requests with a timestamp older than the one in the reservation; for other HTMs, the mechanism should be appropriately adapted.

6.1 Limited fair queuing

The previously described proposal enables any writer to proceed execution after the current holders of the line commit. However, it does not implement any queue for the remaining readers or writers. Once the reservation is served, the rest of the requests will race for the line. An alternative implementation can provide a result equivalent to a limited fair queue.

We make use of the optional read_count field and W flag in the directory. Once the reservation has been set, any GETS request for the same line will be NACKed and the read_count will be incremented. To prevent counting the same read request twice, every request message includes a nacked bit, which is set by the requestor cache on every retry. Only requests with nacked 0 increase the read_count in the directory. On the first GETX request not coming from the original requestor *C*, the W flag is set, and read_count is no longer incremented. This ensures that if the block is requested in exclusive mode during a reservation, the read_count field will contain the count of previous read requests for the block, which will have to be served before acknowledging any exclusive request. Once the original reservation is served, the directory will continue to keep the W flag set, and decrease read_count on every GETS request served, while exclusive requests are NACKed by the directory. When read_count reaches 0 and the W bit is set, only a single GETX request will success (and, in the case of a new conflict, generates a new reservation).

This design does not implement a real queue, given that the directory does not record the identity of read and write requestors. Once the original reservation is served, only the amount of read requests before any write request will be preserved. If new readers try to access the line, nothing prevents them from doing so before the next writer succeeds. If a new writer comes and wins the request race, its request will be satisfied. However, this is enough to assure that the *proportion of sharers and writers* in the queue is satisfied. We consider that this mechanism is fair in that, on average, the waiting time for sharers and writers is the same as it would be with a real queue.

6.2 Thread de-scheduling and migrating

Given that LogTM transactions block accesses that conflict with their read or write sets, thread descheduling is an important issue for HW transactions. Signature-based solutions for this issue have been proposed in [39]. In this sub-section we present how to prevent starvation if the reservation owner gets de-scheduled. If this happened when the resource became free, there would be no request for the line. This would block any other threads trying to access the line. This case does not generate a deadlock but a temporal starvation in the same manner as de-scheduling a thread which is waiting in a lock queue. To cover this last case, in [16] a new mechanism is proposed to detect de-scheduled threads. Waiting threads periodically “publish evidence” that they are still iterating, in the form of a timestamp increase. If other thread finds a timestamp not increased in a long time, it can “jump ahead” the queue.

In our case, we include a counter in the RT line (not depicted in Figure 13). This counter records the current clock on every new GETX request received from the reservation holder. When a conflicting request is received (that should be NACKed by the directory), the counter is checked against the current clock. If the difference exceeds a threshold, the reservation line is cleared to prevent starvation. Such threshold will be set to several times (2 or 3) the delay between requests to cover the case in which network congestion delays a request. A similar case must be considered for readers and writers when using the limited fair queuing proposed in Section 6.1.

6.3 Evaluation

In the multiple-reader, single writer problem, prioritizing readers over writers provides the highest throughput, as writers never stop concurrent readers. Thus, any fairness mechanism will reduce throughput. Similarly, the usage of the Reservation Table can locally introduce more contention in the system as the reservation owner stops furthers threads that could continue execution. On the other hand, starved threads do not contribute to the global throughput, so temporarily blocking other threads to guarantee writer progress might lead to higher throughput. Thus, our implementation should try to produce the lowest performance degradation while still ensuring starvation freedom.

We implemented three versions of the RT mechanism with different reservation policies:

- RT-base: nack'ed requests always make the reservation.
- RT-delayed: the nack'ed request is retried several times (we used 100 in our experiments) before it makes the reservation.
- RT-oldest: the nack'ed request only performs the reservation when the nack'ers have higher (younger) HTM timestamps. This prevents a writer from stopping transactions that started before.

We also compared our proposal with a different conflict resolution policy: the Hybrid model presented in [1] (labelled Abort Policy). Such policy targets the same problem but considering HTM transactions only. With this policy, write requests abort readers that have a higher (younger) timestamp. It requires the HTM to allow transactional requests aborting remote transactions.

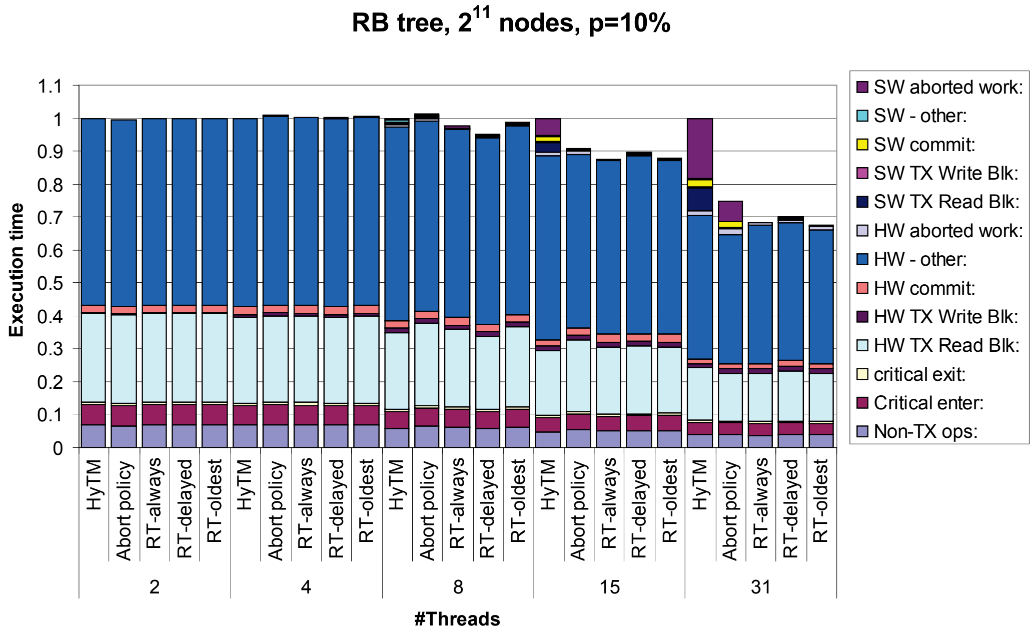


Figure 14: 2^{11} RB-tree execution dissection using different contention mechanisms. p = 10%

Figure 14 Presents a cycle dissection of the average transaction execution of the RB-tree benchmark with size 2^{11} and p = 10%, using 2 to 31 threads (simulating a 32 processor machine in this last case). Transaction execution time is normalized to the base Hybrid system (HyTM). With a small thread count (2 to 4), the system does not suffer from writer starvation and the mechanism does no penalize execution. The contention increases with the thread count, showing a higher percentage of time wasted in SW and HW transaction aborts in HyTM for 15 and 31 threads. The RT mechanisms stop such congestion halting new readers and the performance is significantly increased up to 30.8% with the best RT-oldest policy, but without big differences between them. By contrast, the Abort Policy model is based on aborting running transactions, which increases the abort count and fastens the switch to the slower STM mode, more prone to aborts. This policy also improves the HyTM performance but only up to 25.03%.

Finally, Figure 15 presents other two cases. The left graph shows normalized execution time for a highly contended (p=25%) skip-list with 2^{11} maximum nodes. High thread counts (15 to 31) present much contention, which is clearly avoided by the RT mechanism and not so much by the HyTM policy in the 31-thread case. The right graph shows the Hash-table, which suffers almost no congestion and, therefore, it is not affected by these mechanisms.

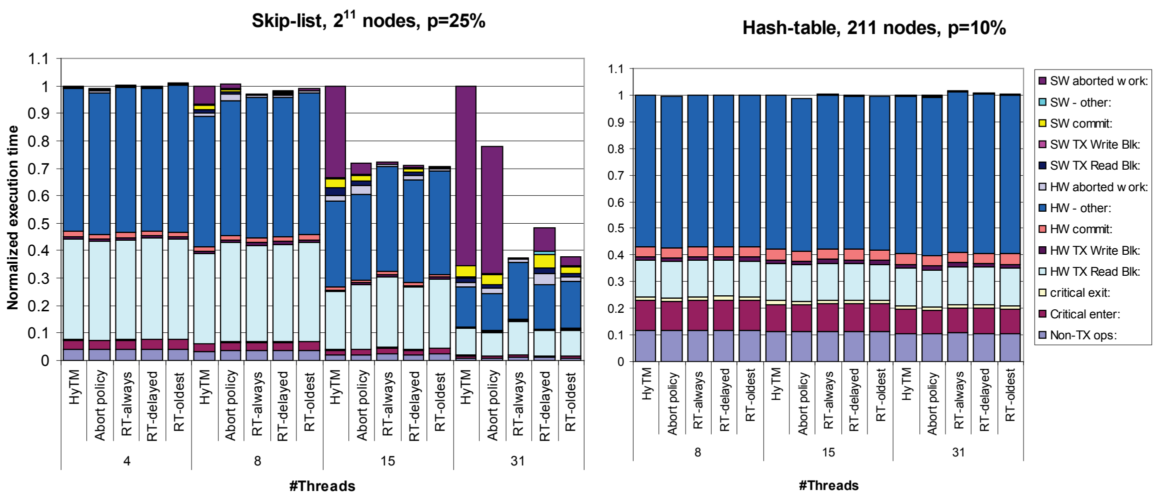


Figure 15: Normalized execution time of Skip-list (p=25%) and Hash-table (p=10%)

7. Related Work

Early STM implementations aimed for providing non-blocking progress guarantees. Ennals [8] argues that lock-based STM implementations are sensible given their integration with the rest of the runtime 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 [15], [23] or in the commit phase [5] to simplify the design and provide higher performance. From these previous works, [5] and [15] can fail in problems such as the privatization one. The work of Menon *el al* [23] covers the problem and proposes several alternative semantics for STM-only safe transactions. However, they rely on whole-program analyses, JIT optimizations and barriers on volatile accesses. Our work is orthogonal to that one, being possible to adapt our HW acceleration proposal to their lock-based system.
Rajwar and Goodman [31], and Martínez and Torrellas [26] 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 [35] extends a traditional HW coherence protocol and cache operations to detect conflicts between transactions. In [32] 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 [4] is the first to consider HW acceleration by a generic HTM system. However, their proposal compiles each transaction's code twice: once for HW supported transactions and the second for SW ones. This system is based on hash-addressed ownership records (*orecs*), and introduces the idea of extending HW transactions to detect conflicts with SW ones by checking the corresponding *orec* status. The improvement in PhTM [20], proposes different transaction execution phases for hybrid systems, such as HARDWARE, HYBRID or SOFTWARE. While we do not consider such phase division, our proposal would be applicable to their HARDWARE and HYBRID phases of execution, affecting *orecs* instead of locks. Even more, such work also introduces the idea of using scalable non-zero indicators instead of counters in the *modeIndicator* variable. This would also suffer from strong starvation in the general case, which could be managed by the Reservation Table. As a note, the authors had to modify the contention manager in LogTM, changing the stalling mechanism addressed in this work. NZTM [38], is another hybrid proposal (also with a modified contention manager) that achieves zero-indirection STM, eliminating some of the performance overheads of our base STM.

In [28], the authors propose a hybrid system based on TL2 [5]. 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.

The LogTM HTM model was presented in [29]. The problem of writer starvation in the LogTM model was discussed in [1], labelled STARVINGWRITER. The authors propose what we called Abort Policy. Our approach is different in that we do not need to abort remote transactions, and we explicitly cover the case of both HW and SW transactions using a HyTM.

Contention management for TM has been previously considered in many different works such as [4] or [33]. However, as far as we know no other work has addressed cache-line fairness issues for directory-based hybrid systems.

Many different HW mechanisms have been proposed to improve the performance of shared-memory synchronization and exclusion. Software reader-writer queue-based locks [27], as the ones used in our base hybrid TM, reduce contention by using a queue of waiters, at the cost of increased memory usage. QOLBY [11], was the first proposal to improve shared-memory synchronization using hardware distributed queues. Memory-side atomic operations, first used in the NYU Ultracomputer [12], perform atomic operations in the memory controller rather than the processors' caches to prevent cache lines bouncing between processors. Recently proposed Active Memory Operations [9] extends the performance to streams of data. Active Messages [6] is a software proposal to move computation to the owner node, considering that the programmer knows where it resides. In forthcoming works of the authors, the extension of Directory Reservations to support fair access for explicit synchronization will be considered. Although possible, we do not know of any of these works specifically addressing writer starvation in shared-memory synchronization. Even more, the complexity of our directory is much lower than any of the previously mentioned mechanisms.

8. Conclusion and future work

This paper has studied the implications of using a pessimistic reader-writer-lock-based transactional memory system with semivisible readers. While this system does not fail in problems such as the privatization one, the performance is seriously degraded by the readers' visibility. A hybrid approach that can use any generic HTM as an acceleration mechanism has been proposed. The overheads incurred by the locks can be easily avoided in atomic blocks, allowing HW and SW transactions to coexist together.

It has been shown that the use of the HyTM can negatively affect some desirable properties of the original STM, specifically the fairness in the resource access. We have shown how, in corner cases, this can lead to program blocking or thread starvation. Such solutions, proposed for HTM systems can be suboptimal for a HyTM model. Directory Reservations is proposed as a low-cost mechanism for the HTM to build a "semi-fair" queue to access blocked resources. This solution removes the starvation problem minimizing aborts and leads to higher performance.

The author's future work includes extending the idea of Directory Reservations to accelerate explicit locking. Having a hardware mechanism implementing this semi-fair queue could offload this task from a software lock, thus reducing the coherence congestion and the lock passing time.

Acknowledgments

This work has been supported by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research and by the Ministry of Science and Technology of Spain under grants TIN2007-60625, TIN2007-6802-C02-01 and CONSOLIDER CSD2007-00050.

References

[1] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift and D. A. Wood, *Performance Pathologies in Hardware Transactional Memory*. International Symposium on Computer Architecture (ISCA), June 2007.

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

[3] 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.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum. *Hybrid transactional memory*. 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct. 2006.

[5] 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.

[6] T. von Eicken, D. Culler, S. Goldstein and K. Schauser. *Active Messages: A mechanism for integrated communication and computation*. In Proc. of the 19th ISCA, May 1992.

[7] F. Ellen, Y. Lev, V. Luchangco and M. Moir, *SNZI: Scalable Non-Zero Indicators*. In Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing. Portland, Oregon, USA, August 12 - 15, 2007.

[8] Robert Ennals Software *Transactional Memory Should Not Be Obstruction-Free*. Intel Research Cambridge Technical Report. IRCTR-06-052.

[9] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim and M. A. Parker. *Active memory operations*. In Proceedings of the 21st Annual international Conference on Supercomputing. Seattle, Washington, June 17 - 21, 2007.

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

[11] J. R. Goodman, M. K. Vernon and P. J. Woest. *Efficient syncrhonization primitives for large-scale cache-coherent multiprocessors*. In Proceedings of the 3rd international conference on Architectural support for programming languages and operating systems (ASPLOS'89), Boston, MA, USA, 1989.

[12] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph and M. Snir. *The NYU multicomputer – designing a MIMD shared-memory parallel machine*. IEEE TOPLAS, 5(2):164–189, Apr. 1983.

[13] 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*. Proc. of the 31st Intl. Symp. on Computer Architecture (ISCA), Munich, June 04.

[14] 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.

[15] 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.

[16] B. He, W. N. Scherer III and M. L. Scott. *Preemption Adaptivity in Time-Published Queue-Based Spin Locks*. 11th Intl. Conf. on High Performance Computing, Dec. 2005.

[17] 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.

[18] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kunduand Anthony Nguyen. *Hybrid Transactional Memory*. In the 11th Symposium on

Principles and Practice of Parallel Programming (PPoPP), New York, NY, Mar. 2006.

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

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

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

[22] 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.

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

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

[25] 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*. Computer Architecture News (CAN), Sept. 2005

[26] 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.

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

[28] C. C. Minh, M. Trautmann, J. W. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis and K. Olukotun. *An Effective Hybrid Transactional*

Memory System with Strong Isolation Guarantees. In 34th International Symposium on Computer Architecture. San Diego, June 2007.

[29] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood. *LogTM: Log-Based Transactional Memory*. Proc. of the 12th Intl. Conference on High-Performance Computer Architecture (HPCA), Austin, TX, Feb. 2006.

[30] 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.

[31] 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.

[32] 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.

[33] W. Scherer and M. Scott. *Advanced contention management for dynamic software transactional memory*. In Proc. 24th Annual ACM Symposium on Principles of Distributed Computing, 2005.

[34] T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore and B. Saha. *Enforcing isolation and ordering in STM*. Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'07). San Diego, California, 2007.

[35] 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.

[36] 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.

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

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

[39] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift and D. A. Wood. *LogTM-SE: Decoupling Hardware Transactional Memory from Caches*. Int. Symposium on High Performance Computer Architecture (HPCA), Feb. 2007.

[40] 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.