Testing Patterns for Software Transactional Memory Engines

João Lourenço
CITI / Departamento de Informática
Universidade Nova de Lisboa, Portugal
joao.lourenco@di.fct.unl.pt

Gonçalo Cunha
CITI / Departamento de Informática
Universidade Nova de Lisboa, Portugal
goncalo.cunha@gmail.com

ABSTRACT

The emergence of multi-core processors is promoting the use of concurrency and multithreading. To raise the abstraction level of synchronization constructs is fundamental to ease the development of concurrent software, and Software Transactional Memory (STM) is a good approach towards such goal. However, execution environment issues such as the processor instruction set, caching policy, and memory model, may have strong influence upon the reliability of STM engines.

This paper addresses the testing of STM engines aiming at improving their reliability and independence from execution environment. From our experience with porting and extending a specific STM engine, we report on some of the bugs found and synthesize some testing patterns that proved to be useful at testing STM engines.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming — *parallel programming*; D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging — *Diagnostics*; D.4.1 [**OPERATING SYSTEMS**]: Process Management — *Synchronization*

General Terms

Algorithms, Performance, Reliability, Experimentation

Keywords

Software Transactional Memory, Testing, Debugging, Testing Patterns, Concurrency

1. INTRODUCTION

With the advent of multi-core microprocessors, multithreading moved from a wanted feature into a vital necessity. Programming with multiple threads forces the programmer to understand and deal with concurrency, and concurrent programming is known to be difficult.

PADTAD'07, July 9, 2007, London, England, United Kingdom.

The usual synchronization constructs (locks and condition variables) while simple on the paper, may become unpredictable and error prone in larger systems. Coarse grained locks, on large data structures, do not scale to large systems and fine grained locks are prone to difficult problems in larger systems, such as deadlocks and priority inversion. Also, locks and condition variables do not compose [5]. An example is a list implementation where all methods (get, add, delete, etc) are internally synchronized with locks. The composability problem appears when the programmer wants to move an object from one list instance into another, without exposing the intermediate state. Having the list implementation to expose the internal lock and unlock methods breaks the list abstraction and introduces deadlock problems.

Software Transactional Memory (STM) may help to solve these problems: it supports composability and is not subject to deadlocks or priority inversion.

The following Section summarizes the main design and implementation approaches to support STM. In Section 3 some of the most difficult bugs we diagnosed are described, and the testing programs used are synthesized into testing patterns and evaluated. We conclude with some more considerations on the work ahead.

2. STM DESIGN AND IMPLEMENTATION OPTIONS

All STM software engines rely on a set of design, implementation and execution environment options, that influence their behavior. Some of such options are analyzed in the following sections.

2.1 Synchronization

STM engines make use of blocking [1,2,3] or non-blocking [4,7,9] techniques to implement synchronization.

Non blocking synchronization techniques make private copies of data as needed and use an atomic compare-and-swap to replace the changed items/data when they are done. Threads that try to update an item that was changed between the first access and the commit, must restart the operation. Blocking synchronization techniques use locks to prevent other threads to access the same data, and release the locks when the operation is complete. Threads that find a locked item wait until the lock is released.

This paper focus on lock-based (with blocking synchronization) STM engines.

2.2 Recovery

When a transaction updates some data variable, the STM system (as with other types of transactional systems) must record in some way the previous value of the variable, in case the transaction does a rollback. Blocking STMs use one of two main recovery strategies:

• Undo log — When a write occurs, the shared variable is locked (to prevent other transactions from accessing the dirty data) and the value is replaced with the new tentative value. The previous value is recorded on a transaction private undo log. This kind of update is called in-place. Reads check the variable lock status, and may access the variable directly only if it is unlocked.
When a commit is performed, the undo-log is discarded and locks are released. When an abort is performed the shared variables are re-written with the values on the undo-log and locks are released. Some authors call this strategy "*encounter mode*".
STM engines operating in undo-log mode include [2, 3, 6].

• Redo log — When a write occurs, the shared variable's value remains unchanged and the tentative value is written on a transaction private write-set. This kind of update is called out-of-place. Reads can't look directly to the shared variable as it may have already been written by the local transaction; instead they must look aside to the transaction write-set. When a commit is performed, the shared variables values are locked and then replaced with the values on the write set. When a rollback is performed only clean-up work needs to be done. Some authors call this strategy "*commit mode*".
STM engines operating in redo-log mode include [1, 2].

2.3 Concurrency Control

Most STM systems use optimistic concurrency control for reads and versioned write locks. Each object has a version number that is incremented for every commit that updates it. When a transactional read is performed the version number of the object is stored in a transaction local data structure. On commit of every transaction, it is verified if the versions of every object read have not changed. If any of the objects has changed, the transaction is in an invalid state and must be rolled back and retried. This prevents non-serializable orderings from being committed.

This concurrency control approach does not preclude some other execution hazards. While the transaction is being run, it may see an incoherent state as a result of an unfortunate interleaving. As a result it may enter an endless loop or dereference invalid pointers. Such transactions are invalid and must be aborted. The transactional engine may detect this incoherence by re-validating the whole read set at every transactional load, periodically or at commit time. Figure 1 shows an example of a transaction observing an invalid state and, consequently, entering an infinite loop.

TL2 [1] uses a slightly different algorithm to avoid this problem. It uses a global version clock instead of a version counter per variable. When a transaction starts, it reads the global clock. When a transaction commits, the written variables are updated to the current number of the global clock. On a transaction load, the transactional engine checks whether the current version of the variable is greater than the value read on the beginning of the transaction. If it is the transaction immediately aborts. With this algorithm TL2 prevents transactions from running in inconsistent states.

	T1	T2
1	*// x=0 && y=0*	*// x=0 && y=0*
2	**atomic** {	
3	a=x;	
4	.....	.....
5		**atomic** {
6		x++;
7		y++;
8		}
9	.....	.....
10	b=y;	
11	**if** (a!=b)	
12	**while** (true) {}	
13	}	

Figure 1: Infinite loop caused by improper ordering

2.4 Transactional Granularity

Transactional shared variables may have different levels of granularity. Granularity may be at the object level (object-based STMs [7]) or block level (word-based STMs [4]). With block level granularity, the block may have the size of a memory word or the size of a cache line.

In object based STMs, all the object fields share the same metadata—they have a coarser granularity than word based STMs. Concurrent accesses from different transactions to different fields of the same objects will conflict. The transactional API must be called once for every object.

Word based STMs have more room for concurrency as the granularity is finer, each word has its own metadata, and collisions are less frequent. However the overhead is also higher, as the transactional API must be called once for every field the transaction accesses.

Object based STMs are less suitable for accesses to transactional variables that are not objects—like single variables and arrays. Word based STMs, on the other hand, can access single variables and access arrays word-by-word but incur in higher overhead.

2.5 Lock Placement

There are two main strategies for lock placement on lock based STMs. One is to place the lock next to the data. The other is to place the lock on a lock separate table.

2.5.1 Lock Adjacent to Data

Placing the lock adjacent to the data has the advantage of having a higher chance that both, the lock and the data, stay in the same cache line. This may yield to a better performance by reducing the number of cache misses.

However, placing the lock and data near to each other, requires changes to the internal representation/structure of the objects in the heap and in the way they are handled. Frequently, a pointer to the object points to the header metadata, and to access the object itself the pointer must be explicitly incremented by the header size. This approach usually limits the re-usage of existing libraries.

STMs that place locks adjacent to data include [1, 2, 3, 6] place locks adjacent to data.

2.5.2 Lock Table

Placing the locks in a separate table does not require changes to the object's internal representation/structure,

nor in the way objects are handled. This technique makes it is easier to re-use existing libraries, as it only requires the memory accesses (reads/writes) to be instrumented.

To map an object to the lock in the lock table, it is necessary to have a hash function that maps the object's address to a lock table entry.

STMs that use a separate lock table to store locks include [1, 2].

3. TESTING STM IMPLEMENTATIONS

From our experiments aiming at porting, extending and testing a specific STM engine, we have synthesized some testing patterns which increase the probability of triggering wrong (buggy) behaviors.

3.1 Experiment Setup

Our work was based on TL2 implementation [1]. TL2 is word based STM with a redo-log strategy. TL2 has the option of keeping locks adjacent to the data or in separate lock table. It uses optimistic reads and versioned write locks, with a global version clock algorithm to prevent transactions from running in inconsistent states.

TL2 original source code was developed with the SUNPRO C compiler, for SPARC machines and the SOLARIS operating system. Our prototype began with porting TL2 to compile with the GCC compiler, and targeting Linux systems with Intel x86_32 and x86_64 microprocessor architectures. Afterwards, several modifications were made to allow the usage of both, object- and word-based modes, with undo- and redo-log strategies. These were core changes to the STM engine, which caused several bugs to show up.

3.2 Terminology

Before showing a sample of the problems found during the development of our version of the STM engine, we describe the terminology used in the examples. Figure 2 describes the operations made by the transactions at the transactional level: start, load, store, commit, and abort. Figure 3 describes the operations made by the STM engine at the lock and data level: read variable's lock version, read variable's value, etc.

Symbol	Meaning
$TxStart()$	Start transaction
$TxCommit()$	Commit transaction
$TxAbort()$	Abort transaction
$TxLoad(x)$	Transactional operation to read the value of variable x
$TxStore(x, a)$	Transactional operation to write the value a to variable x

Figure 2: Transaction Construct Glossary—Transactional Level

Figure 4 shows a simplified transformation of transactional-level operations (listed in 2) into lock-level ones (listed in 3). With "simplified" we mean that some internal operations, like adding elements to read and write sets or internal verification and maintenance operations, were omitted in the transformation. The omitted operations are not relevant to the illustration of the bugs reported in this paper.

Symbol	Meaning
$a = RV(x)$	Read the value of transactional variable x
$v = RL(x)$	Read the lock version of the transactional variable x
$WV(x, a)$	Write the value a to the transactional variable x
$Acq(x)$	Acquire the lock of the transactional variable x
$Rel(x)$	Release the lock of the transactional variable x

Figure 3: Transaction Construct Glossary—Lock and Data Level

Operation	Decomposition for a redo-log STM	Decomposition for a undo-log STM
$TxStart()$	$ts = clock;$	$ts = clock;$
$a = TxLoad(y)$	$v1 = RL(y);$ $a = RV(y);$ $v2 = RL(y);$	$v1 = RL(y);$ $a = RV(y);$ $v2 = RL(y);$
$TxStore(x, a)$		$Acq(x);$ $WV(x, a);$
$TxCommit()$	$Acq(x);$ $RL(y);$ $WV(x, a);$ $Rel(x);$	$RL(y);$ $Rel(x);$
$TxAbort()$		$WV(x, old);$ $Rel(x);$

Figure 4: Simplified decomposition of transactional- into lock-level operations in undo- and redo-log mode STMs

TxStart —In both, redo- and undo-log based STMs, a timestamp will be associated to the transaction. The timestamp will be the value of the current global version clock.

TxLoad —In both, redo- and undo-log based STMs, loading a variable (memory location) is, basically, a three-step operation: i) load the variable's version counter; ii) load the variable's value; iii) load again the variable's version counter. After these steps it is checked whether that variable's version number hasn't changed between the first and the third step. If it did the transactions must abort immediately, otherwise the address is added to the read set. These checks are not relevant for the bugs analyzed in this paper and were, therefore, omitted in the decomposition of the operation.

TxStore —In redo-log mode, changes are made out-of-place. The new variable's value will be stored in the redo

log, and no memory changes are actually made at this point. The new value will only overwrite the original during transaction commit, if it succeeds. If commit fails then the redo-log is simply discarded.

In undo-log mode, changes are made in-place. Once the lock of the variable has been acquired, the current variable's value will be stored in the undo-log and then overwritten with the new value.

TxCommit —In redo-log mode, all the pending updates should now become effective. In this case, commit is a four-steps operation: i) acquire (write) locks for all variables that will be updated; ii) validate the transaction read-set. Validating the read-set ensure that all variables read by the transaction satisfy two conditions: they are not currently locked by any other transaction; and they were not changed since the moment they were first read until all the write locks have been acquired; iii) apply all pending changes to memory locations (overwrite the memory locations with the values kept in the redo-log); iv) release all acquired locks, increasing the version number.

In undo-log mode, the locks overwritten variables were already acquired in the $TxStore$ operation and new values written into variables. It is only necessary to validate the read-set and, if successful, release all acquired locks, increasing their version number.

TxAbort —In redo-log mode, abort simply discards the redo-log, so it is a null operation in what concerns to changes to shared memory locations.

In undo-log mode, new values were written in-place. So, when aborting a transaction, the original variable values must be restored from the undo-log, and the previously acquired locks released.

TxSterilize —This function is called before releasing any transactional variable. Both in undo and redo log mode, it prevents all transactions from doing any further reads or writes to that variable. It does so by increasing the lock version of the variable to the current global clock number.

3.3 Sample of Bugs Found

In the following section we describe interleavings that triggered wrong behaviors in the STM.

3.3.1 Bug 1: Reference to Non-Transactional Memory

Figure 5 shows a problem where transaction T1 is accessing a piece of transactional memory already released by another transaction T2. The transactions are operating on a list with three nodes: x, y, and z. T1 is iterating the list reading the nodes keys and T2 is deleting node y from the list.

The bug arises when T1 does not detect that its read set is not consistent when performing step 8. In such case, it will run in inconsistent state, as the memory freed in step 7 may have already been recycled for other uses.

The original *TxSterilize* function only waited for all writes to variable y to drain, allowing this harmful interleaving. To correct the problem, *TxSterilize* was made to also increment the version lock of variable y to the current value of the

	T1	T2	Description
1	$y = TxLoad(x.n)$		get the pointer to node y
	.....	.....	.....
2		$y = TxLoad(x.n)$	
3		$z = TxLoad(y.n)$	
4		$TxStore(x.n, z)$	
5		$TxCommit()$	
6		$TxSterilize(y)$	
7		$free(y)$	
	.....	.....	.....
8	$TxLoad(y.k)$		access to freed memory

Figure 5: Undo/Redo-log mode: Reference to Non-Transactional Memory.

```
for each i in write-set {
  if (i is not locked){
    if(i is also in read set){
      // read/write variable
      if(get_lock_version(i) > tx_timstamp)
        abort;
      else
        lock(i)
    }else{
      // write only variable
      lock(i)
    }
  }
}
```

Figure 6: Lock acquisition in redo-log mode—buggy version

global clock. In this way, transactions that accesses y after the sterilization will detect that the version clock has been updated and will, therefore, abort.

3.3.2 Bug 2: Lost Update with Small Lock Table

As depicted in Figure 6, commit starts by searching every variable in the write set. If the variable is also in the read set then it's a read/write variable, otherwise it's a write only variable. For read/write variables, the lock version is checked if it is greater than the transaction timestamp; for write only variables, the algorithm didn't find such check necessary.

Figure 7 shows a possible interleaving on two transactions T1 and T2. T1 is updating the write-only variable y and the read/write variable x. T2 is only updating x.

When operating in redo-log mode, the original algorithm didn't work with the interleaving shown in Figure 7, which includes a lock collision. If variable x and y have identical hashes, there will be a lock collision and they will share the same lock in the lock table. With this interleaving, when T1 commits, the lock acquisition of variable y will be successful, because it is a write only variable, and the lock version validation of variable x will be skipped, because the lock was already acquired.

The correction to this problem is to always validate the lock version of read/write variables, even if the lock is al-

	T1	T2	Description
1	$TxLoad(x)$		
	.....	.....	.....
2		$TxStore(x, a)$	
3		$TxCommit()$	
	.....	.....	.....
4	$TxStore(y, a)$		y is write only
5	$TxStore(x, a)$		x is read-write
6	$TxCommit()$		
7	$\hookrightarrow Acq(y)$		lock acquisition phase
8	$\hookrightarrow Acq(x)$		
9	$\hookrightarrow RL(x)$		read set validation phase

Figure 7: Redo-log mode: Lost Update with Small Lock Table

```
for each i in write-set {
  if(i is also in read set){
    // read/write variable
    if (i is not locked &&
        get_lock_version(i) <= tx_timstamp) {
      lock(i);
      }
    else if(i is locked by this thread &&
      get_lock_version(i) <= tx_timstamp)
      continue;
    else
      abort();
  }else{
    // write only variable
    lock(i);
  }
}
```

Figure 8: Lock acquisition in redo-log mode—correct version

ready held. Figure 8 shows a possible implementation for the corrected algorithm.

3.3.3 Bug 3: Dirty Read Not Invalidated when Transaction Aborts

Figure 9 shows an example where transaction T1 reads the variable x and commits, and transaction T2 writes to the same variable and aborts. If the illustrated interleaving occurs when operating in undo-log mode, the locks are immediately acquired during the *TxStore* and the previous value of x is stored in the undo log. In step 2, T1 reads the lock version of variable x. Then T2 stores a new value on x. In step 6, T1 (dirty-)reads the value of x after changed by T2. In step 7, T2 aborts and the old value of x is restored.

The bug in this situation was that, when transaction T2 aborted (in undo-log mode) the value of variable x was restored and the lock was simply being released with the old version number. Transaction T1 was not detecting that it read a dirty value because the lock version revalidation, in step 10, returned the same value as in step 2, therefore assuming the value read in step 6 was valid. To correct this problem, when aborting a transaction in undo-log mode, the lock version of every variable in the write set must be incremented.

	T1	T2	Description
1	$TxLoad(x)$		T1 loading variable x.
2	$\hookrightarrow RL(x)$		
	.....	.....	.....
3		$TxStore(x, a)$	
4		$\hookrightarrow Acq(x)$	
5		$\hookrightarrow WV(x, a)$	new value is written
	.....	.....	.....
6	$\hookrightarrow RV(x)$		dirty value is read by T1
	.....	.....	.....
7		$TxAbort()$	T2 aborts
8		$\hookrightarrow WV(x)$	old x value is restored
9		$\hookrightarrow Rel(x)$	
	.....	.....	.....
10	$\hookrightarrow RL(x)$		lock version revalidation
11	$TxCommit()$		

Figure 9: Undo-log mode: Dirty read not invalidated when transaction aborts

3.3.4 Bug 4: Lost Update on Lock Upgrade

Figure 10 shows a problem that happened on undo-log mode, when reading a variable and then modifying its value. Transaction T1 is incrementing the variable x and transaction T2 is storing a new value in the same variable. The problem was that *TxStore* was not validating if the lock version was the same as the one obtained in the first *TxLoad*—it was merely acquiring the lock and writing the value. The correction was to force *TxStore* to validate the lock version before writing to the variable.

	T1	T2	Description
1	$v = TxLoad(x)$		
	.....	.....	.....
2		$TxStore(x, w)$	
3		$TxCommit()$	
	.....	.....	.....
4	$TxStore(x, v + 1)$		upgrade from read access to write access

Figure 10: Undo-log mode: Lost update on lock upgrade

3.4 Testing Patterns

Testing the STM engine aims at incrementing the probability of generating harmful interleavings. Harmful interleavings are those that improperly read and/or modify the shared data, i.e., locks and transactional variables. Such interleavings can occur during reads, writes/updates, commits, aborts, and when adding and removing variables from the transactional space.

Tests may target specific implementation options, such as the bug described in Section 3.3.2 or concurrency control errors, which depend on both implementation options and execution environment. For concurrency control errors, we want to maximize the function $f_R = \sum_i \frac{C_i}{T_i}$ where C_i is the

time a transaction i runs with the shared state changed and T_i is the total run-time for the same transaction.

3.4.1 Very Short Transactions with Read/Write Operations

This testing pattern aims at maximizing the interleavings between the main transactional operations, i.e., reads, writes, commits and aborts. This test also maximizes the frequency of commits.

This pattern is very adequate for redo-log based STM engines. Such systems only change the shared state on commit, shortening the time-window in which the transaction runs with its shared state changed and concurrency errors can only be revealed once there are changes in the shared state.

This testing pattern was found useful at finding the bug reported in Section 3.3.1. Since the bug only occurred when releasing transactional memory and the release is made after the commit, a higher number of commits maximizes the occurrence of the bug.

3.4.2 High Frequency of Variables Being Added and Deleted from Transactional Space

A testing pattern aiming at stressing the variability of the transactional space, by the repeatedly insertion and removal of data to/from the transactional space.

Such pattern allows to detect bugs mostly related to transactions holding pointers to variables being simultaneously released by other transactions, such as the bug described in Section 3.3.1. These bugs may cause invalid memory accesses and memory being read/written after data deletion.

3.4.3 High Number of Updates on a Small Number of Variables

This testing pattern aims at generating a very high frequency of collisions between transactional reads and writes, also forcing transactions to abort very frequently. This pattern can easily be instantiated with a list that can hold a very small amount of nodes, e.g., ten or less, and with several transactions trying to add/update/delete the list nodes.

This test produced very good results with undo-log based STM engines, because they change the shared data on writes, changing the locks and data values, on commits, changing only the locks, and on aborts, changing the locks and data values. Overall, this testing pattern was found to be one of the most effectives.

This pattern was particularly useful to find the bugs reported in Sections 3.3.2 and 3.3.3, as these bugs are related to read/write collisions.

3.4.4 Small Lock Table

STM engines that use a lock table to store objects/data locks, usually make use of an hashing function to map the object address to its lock within the lock table. Such hashing function may map several objects to the same table position, originating a lock collision problem. Such lock collision may cause an improper validation of the lock state, with transactions never being able to commit and potentially running into livelocks.

This pattern aims at maximizing the function $f_L = \frac{V*T}{L}$, where V is the average number of transactional variables being manipulated by each transaction, T is the number of running transactions and L is the size of (number of entries in) the lock table. The pattern can, therefore, be instantiated by an adequate combination of i) using a smaller lock table; ii) increasing the number of transactional variables; and iii) increasing the number of running transactions.

This pattern contributed significantly to find the bug reported in Section 3.3.4. Just by decreasing the size of the lock table to a very small number it was possible have a significant number of lock collisions and reproduce the harmful interleaving.

3.4.5 More Concurrent Transactions than CPUs

If the number of transactions is less than the number of CPUs, any transactions willing to run can be immediately assigned to a CPU, and transactions will never be stalled waiting for CPU. In such case, some interleavings will be harder to reproduce, because they depend on transactions being preempted and stalled for some time.

This pattern was useful to reproduce the bug reported in Section 3.3.3. This bug depends on the transaction T1 being preempted while executing the *TxLoad* operation.

4. CONCLUSIONS

We have elaborated on some of the most relevant options taken while implementing a STM engine. From our work with porting the TL2 engine to the Intel IA_32 and AMD64 processor architectures, and with validation the original TL2 implementation and our own port, we have synthesized a set of testing patterns which fulfilled well their goal: the validation of the STM engine. For behavior cross-referencing, we applied some our tests to other STM engines, with emphasis LibLTX [3].

Our experiments suggest that some of the bugs found in STM engines are directly or indirectly related to out-of-order instruction execution hazards. They also suggest that the execution environment has very strong implications on the STM engine stability, making tests that were running for hours or days without errors to fail in seconds. Tests made on multicore computers may have substantially different results than tests made on multiprocessor computers. Multiprocessors have one cache per CPU, while multicore computers have one cache shared among all cores. On multiprocessors, since there is no cache sharing, a variable may be in one CPU cache but not in another. This leads to a higher frequency of out-of-order executions, supported by the generality of the current processors to improve performance [10].

Much work lays ahead of us concerning multiple facets of STM. Deeper analysis of the behavior patterns subsumed in our testing units is mandatory, as well as having a better understanding on when, how and why a STM engine may fails. To our best knowledge, Manovit et al [8] is the only other published work involving testing STM engines. They get "inspiration" from the analysis of algorithms originally developed for checking traditional memory consistency and propose an axiomatic framework to model the formal specification of a realistic transactional memory system which may contain a mix of transactional and non-transactional operations. This approach is much different from the work reported in this paper.

It is our belief that testing packages should be annotated with invariant tests. Such invariant tests should optionally be verified at run-time, so that the errors could be detected as early as possible, minimize Byzantine effects. Once an error is detected, other tools, such as trace generators and

analyzers and core dump debuggers, should be used to locate, isolate, refine and correct the error cause.

We felt a desperate need of libraries and tools providing low-intrusion logging services for STM based applications and STM engine behavior, of STM specific trace analyzers and of source-level debuggers incorporating the concept of STM.

Acknowledgments

We are thankful to Dave Dice, Ori Shalev and Nir Shavit, for making the TL2 source code available to us.

5. REFERENCES

[1] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

[2] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[3] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[4] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[5] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[6] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[7] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, july 2003.

[8] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143, New York, NY, USA, 2006. ACM Press.

[9] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[10] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25, 1967.