# Software Transactional Memory

*Final Project Report, Multicore Programming, Spring 2012*

Siyuan Zhou <siyuan.zhou@nyu.edu>, Zhiqiang Shen <zshen@nyu.edu>

## Table of Contents

# Introduction

(Siyuan)

The advent of multicore processors has renewed interest in the idea of incorporating transactions into the programming model used to write parallel programs. This approach, known as transactional memory, offers an alternative, and hopefully better, way to coordinate concurrent threads. In this report, we describe this emerging alternative programming model, focusing on the software solution. First, we discuss the reason why we need Software Transactional Memory (STM) and what it offers to programmers. Then we take a specific recent STM algorithm in RSTM suit as an example to demonstrate a practical implementation and explore the consideration and trade-off when building STM system. Beyond that, we take experiments based on RSTM to show the performance and benefits of STM. We close this report with the conclusion that some problem are still open while transactional memory is a promising programming model in the parallel era.

# Background

(Siyuan)

Programmers need a new programming model since multicore architecture has been mainstream and pervasive, because parallel programming poses many new challenges to the developer, one of which is synchronizing concurrent access to shared memory by multiple threads. To explain why we need Transactional Memory (TM), we review and analyze the strengths and weaknesses of the standard synchronization primitives [1].

## Locking

Programmers have traditionally used locks for synchronization, but locking, as a synchronization discipline, has many pitfalls for inexperienced programmers. For example, *Priority inversion* occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads. *Deadlock* can occur if threads attempt to lock the same objects in different orders. The heart of the problem is that no one really knows how to organize and maintain large systems that rely on locking. Reasoning and maintaining correctness with locking is usually hard.

## Atomic Operation

One way to bypass the problems of locking is to rely on atomic primitives like CompareAndSet(). Many non-blocking data structure and algorithm have been researched for their potential performance improvement and scalability. While they are often hard to devise, and sometimes, though not always, have a high overhead. The principal difficulty is that nearly all atomic primitives such as CompareAndSet() operate only on a single word, and programmers have to take snapshot manually to make sure they have a consistent state, which is error-prone and often forces a complex and unnatural structure or algorithms.

## Composition and Abstraction

All the standard synchronization mechanisms have a major drawback: they cannot easily be *composed*. Image that we want to dequeue an item *x* from queue Q1 and enqueue it at another one Q2 *atomically*: no concurrent thread should observe either that *x* has vanished, or that it is present in both queues. Even we have thread-safe Queue implementations, each method acquires the lock internally, so it is essentially impossible to combine two method calls in this way. We can make the internal lock public and lock them outside these queues, like what we usually do before traversing Java concurrent containers. However this approach breaks the abstraction and modularity we build.

## Bug Avoidance

Beyond making concurrent programming easier for programmers, it has been shown in a comprehensive real world concurrency bug characteristic study [2], involving MySQL, Apache, Mozilla and OpenOffice, that Transactional Memory can help avoid many concurrency bugs (41 out of the 105 concurrency bugs). TM can potentially help avoid many concurrency bugs (44 out of the 105 concurrency bugs), if some concerns can be addressed. Although TM is not a panacea, it can ease programmers correctly expressing their synchronization intentions in many cases, and help avoid a big portion of concurrency bugs.

## Why Software Solution

Another approach to Transactional Memory is Hardware Transactional Memory (HTM). HTM research investigates changes to a computer system, such as CPU and cache, and instruction set architecture to support transactions. Usually hardware implementation or support offers higher performance, comparing with software TM, but even hardware TM will eventually be seen as desirable in multi-core machines, STM is still useful for latency systems and as a fall-back when hardware resources are exhausted [3].

# STM Semantics

(Siyuan)

In a parallel program, there are two basic concurrency control problems: mutual exclusion and synchronization. As a new alternative programming model, Transactional Memory has to address these two problems by providing appropriate declaration and semantics [4].

## Atomic Block

### Conflict

Before introducing the semantics of transaction, let us take a look close at the nature of mutual exclusion problem. In parallel programs, a *conflict* occurs when concurrently executing threads access the same shared data and at least one thread modifies the data. When the conflicting accesses are not synchronized, a *data race* occurs. So generally programmers should be able to express their intentions with Transactional Memory to avoid data race easily.

### Declaration

The atomic statement delimits a block of code that should execute in a transaction:

```
atomic {
  x = Q1.dequeue();
  Q2.enqueue(x);
}
```

The block executes with the failure atomicity and isolation properties of a transaction. Failure atomicity means a transaction execute to completion or, in case of failure, to appear not to have executed at all. An aborted transaction should have no side effects. Isolation requires that execution of a transaction does not affect the result of concurrently executing transactions.

From the programmer's perspective, an terminated atomic block has two possible outcomes. If the transaction *commits,* its results become part of the program's state visible to code executed outside the atomic block. If the transaction *aborts,* it leaves the program's state unchanged.

A key advantage of transactions is that an atomic block does not name shared resources, either data or synchronization mechanisms [4]. This feature distinguishes it from earlier programming constructs, such as monitors, in which a programmer explicitly names the data protected by a critical section. It also distinguishes atomic blocks from lock-based synchronization, in which a programmer explicitly names the synchronization protecting data.

### From Implementation View

In implementation, TM system consists four functions: *TMBegin* and *TMEnd* mark the boundaries of a lexically scoped transaction. Remarkably, the result transaction is dynamically scoped. It encompasses all code executed while control is in the atomic block, regardless of whether the code itself is lexicographically enclosed by the block. So, in the above example, the code in function *enqueue* and *dequeue* also execute transactionally.

*TMRead* and *TMWrite* are used to read and write shared locations, respectively. It is necessary to wrap normal read and write to make them transactionally, which could be done by compilers.

*TMEnd* also means the transaction should *validate* its snapshot to make sure it has a consistent state and it is safe to *commit* the transaction. After *commit*, the transaction will be a part of the program state. Besides, *validation* may occur many times in a transaction, for example, after the read from memory in every *TMRead* to detect conflicts, like that in NOrec discussed in the implementation section.

## Synchronization

### Retry

A transaction that executes a retry statement aborts and then reexecutes. This is the programmer-controlled analogue of the actions that occur when most TM systems detect a conflict. Retry, or Self-abort, is a general mechanism that allows a transaction to abandon its current computation for an arbitrary reason and to reexecute in hope of producing a different

result. Unlike the explicit signaling, with conditional variables and signal/wait operations, or the implicit signaling, with a waituntil predicate statement, used in monitors, retry does not name either the transaction being coordinated with or the shared locations. An example is following.

```
atomic {
  if (buffer.isEmpty()) retry;
  x = buffer.dequeue();
}
```

It is suggested delaying reexecution until the system detects changes in one or more of the values that the transaction read in its previous execution, so its outcome may be different.

### OrElse

OrElse make it possible to execute a code block in a transaction, if it aborts, then try the other block, if it fails too, the transaction fails. OrElse makes composition much easier, as demonstrated as follow. The code only blocks if both are empty, which could not be expressed in standard synchronization primitives.

```
atomic {
  {  x = Q1.dequeue();  }
  orElse
  {  x = Q2.dequeue();  }
}
```

## An Instance, RSTM

(Zhiqiang)

Let us introduce RSTM to demonstrate an example of STM semantics. RSTM supports two APIs, C++ STM API and Library API. C++ STM API follows the rules in the draft C++ TM specification, by enclosing atomic blocks with keyword of *__transaction*, and marking transactional functions as *transaction_safe*.

The Library API provides maximum flexibility, with which it is programmer's responsibility to guarantee that any shared memory access is safe. A programmer is also responsible for ensuring that allocation and reclamation are safe, and for using API calls only when it is necessary for the sake of performance.

The following is a simple counter example, which demonstrates these two style of API. With C++ STM API, it looks like:

```
__transaction_atomic { a++; }
```

With the Library API, it becomes:

```
TM_BEGIN(atomic) {
  TM_WRITE(x, TM_READ(x)+1);
} TM_END;
```

As we can see, we utilities macros to mark transactional memory access explicitly. Also It is necessary to include <*api/api.hpp*> before the declaration.

# Implementation

(Siyuan)

We take the default algorithm NOrec [5] in RSTM suit as an example to demonstrate STM implementation details for three reasons.

1. The name NOrec means there is no Owner Record in the algorithm, which is often used in traditional STM algorithms to maintain a transaction's ownership of an object/word. Thus the algorithm is relatively clear and simple without complicated data structures.
2. While NOrec is not complicated, it demonstrates common technologies in STM, such as versioning and snapshot, conflict detection, redo log, read/write set and validation etc. Also it reflects recent STM research trend, lazy conflict detection as an example. So it is an appropriate example for beginners.
3. It is the default algorithm in RSTM suit because of its higher performance, compared to previous algorithms, which is the result of the simple design.

Due to the length of this report, we focus on the mutual execution rather than other STM features, since it is the key component of a STM.

## Design

### Single Global Sequence Lock

A sequence lock [6] resembles a reader-writer lock in which a reader can upgrade to writer status at any time, but must be prepared to restart its critical section if one of its peers upgrades. In implementation, a sequence lock is an unsigned int. When odd, it indicates that a writer is active; when even, zero or more readers may be active. If a reader find it is odd, the reader should restart its critical section to avoid access of the inconsistent state when a writer updating. While if a reader find even, it is safe to read. A writer should use CAS to acquire the lock by increasing it to odd, if it fails there is another writer concurrently, so it restarts. After a writer CAS successfully, it gets the lock and performs writes. Then the writer release the lock by increase it to even. If write is infrequent, sequence lock can enable more concurrent read with few overhead compared to reader-writer lock.
NOrec use a single global sequence lock to protect the transaction commit protocol. Only commit is protected and the validation for commit occurs before lock acquisition, to minimize the time of holding this sequence lock, thus alleviating commit bottleneck. Besides, the global sequence lock provides a natural "consistent snapshot" capability for validation.

### Lazy Conflict Detection/Redo Log

The algorithm uses lazy conflict detection and a redo log for concurrent speculative writers. Updates are buffered in a write log, which we must search on each read to satisfy possible read-after-write hazards. It use a linear write log indexed by a linear-probe hash table with versioned buckets to support O(1) clearing. Writing transactions do not attempt to acquire the sequence lock until their commit points, allowing speculative readers and writers to proceed concurrently.

### Value-Based Validation

Rather than logging the address of an ownership record, a Value-Based Validation (VBV) read barrier (*TMread*) logs the address of the location and the value read. Validation consists of re-reading the addresses and verifying that there exists a time (namely now) at which all of the transaction's reads could have occurred atomically.

Lazy conflict detection, buffered updates, and VBV allow active transactions to "survive" through a non-conflicting writer's commit. This adds significant scalability to NOrec in workloads where writers are common or transactions are long.

## Implementation Details

### Metadata

```
volatile unsigned global_lock
local unsigned lock_snapshot
local List <Address, Value> reads
local Hash<Address, Value> writes
```

The sequence lock is simply a shared unsigned integer. Each transaction maintains a thread local snapshot of the lock, as well as a list of address/value pairs for a read log, and a hashtable representation of a write set.

### Validation

```
unsigned Validate () {
  while (true) {
    time = global_lock
    if ((time & 1) != 0) continue

    for each (addr, val) in reads {
      if (*addr != val) TXAbort() // abort will longjmp
    }

    if (time == global_lock) return time
  }
}
```

Validation is a simple consistent snapshot algorithm. We start by reading the global lock's

version number, spinning if there is a writer currently holding the lock. Then loop through the read log, verifying that locations still contain the values seen by earlier reads. Checking global lock verifies that validation occurred without interference by a committing writer, restarting the validation if this is not true.
The Validate routine returns the time at which the validation succeeded. This time is used by the calling transaction to update its snapshot value.

### Transaction Begin

```
void TMBegin() {
  do
    snapshot = global_lock
  while ((snapshot & 1) != 0)
}
```

Beginning a transaction in NOrec simply entails reading the sequence lock, spinning if it is currently held by a committing writer. This snapshot value indicates the most recent time at which the transaction was known to be consistent.

### Read Barrier

```
Value TMRead(Address addr) {
  if (writes.contains(addr)) return writes[addr]

  val = *addr
  while (snapshot != global_lock) {
    snapshot = Validate()
    val = *addr
  }

  reads.append(address, value)
  return val
}
```

Given lazy conflict detection and buffered updates, the read barrier first checks if we have already written this location. If not, we read a value from memory. then compares the sequence lock to the local snapshot. If the snapshot is out of date, we validates to confirm that the transaction is still consistent, capturing the returned time as the new local snapshot. In the loop, we reread the memory location and may need to try again. The last two lines log the address/ value pair for future validation, and return the value read.

### Write Barrier

```
void TMWrite(Address addr, Value val) {
  writes [addr] = val
}
```

The write barrier simply logs the value written using a simple hash-based set.

### Transaction Commit

```
void TMCommit() {
  if (read-only transaction) return

  while (!CAS(&global_lock, snapshot, snapshot + 1)) {
    snapshot = Validate()
  }

  for each (addr, val) in writes {
    *addr = val
  }

  global_lock = snapshot + 2 // one more than CAS above
}
```

All transactions enter their commit protocol with a snapshot of the sequence lock, and are guaranteed, due to post-validation in the read barrier, to have been consistent as of that snapshot. A read-only transaction linearizes at the last time that it was proven consistent, i.e., snapshot time. No additional work is required at commit for such transactions.

A writer transaction will attempt to atomically increment the sequence lock using a compare-and-swap (CAS) instruction, using its snapshot time as the expected prior value. If this CAS succeeds, then the writer cannot have been invalidated by a second writer: no further validation is required. A failed CAS indicates a need for validation because a concurrent writer committed. The call to “Validate()” performs this validation and moves the snapshot forward, preparing the writer for another commit attempt.

## Limitations

The sequence lock provides for only a single active committing writer at a time. This is the limitation that would likely make it unsuitable as the primary TM mechanism on a machine with hundreds of cores.

# Experiments

(Zhiqiang)
In this paper, we will focus on two kinds of performance benchmark to experiments with RSTM, overhead and scalability. We use the Red Black Tree with 256 nodes as our testing data

structure and three common operations on them, lookup, insert and remove. We simulate two workload behavior scenarios, 50% reads and 98% reads for each software transactional memory algorithms we testing.

## Setup

All the tests is performs on cuda1.cims.nyu.edu machine. The cuda1 comes with two socket CPUs, 4 cores per CPU, and 2 threads per core; that is total of 16 concurrent hardware contexts. RSTM library and benchmarks were compiled with gcc 4.4.6 using -O3 settings and built-in thread-local storage. The program is locate at /tmp/rstm_build on cuda1, and raw benchmark results located at goo.gl/maFoQ

## Overhead

We measure the overhead using single-threaded 256 nodes BRTree executions in two different workloads, 50% reads and 98% reads, and compare two algorithms, the CGL and NOrec. The BRTree with 256 nodes is relatively small result in small transactions, with relatively small write sets due to rotation. Results for the RBTree benchmark are the average of five runs, where each run consists of a single thread executing million pseudo-random transactions obtained from the same initial seed, thus all test do the same amount of useful work.

The CGL algorithm we compared to NOrec is pretty simple, all the transactions acquire and release a single global lock, and the overhead of CGL is one atomic compare-and-swap per transaction, which can be nontrivial for small transactions, but is amortized in larger transactions.

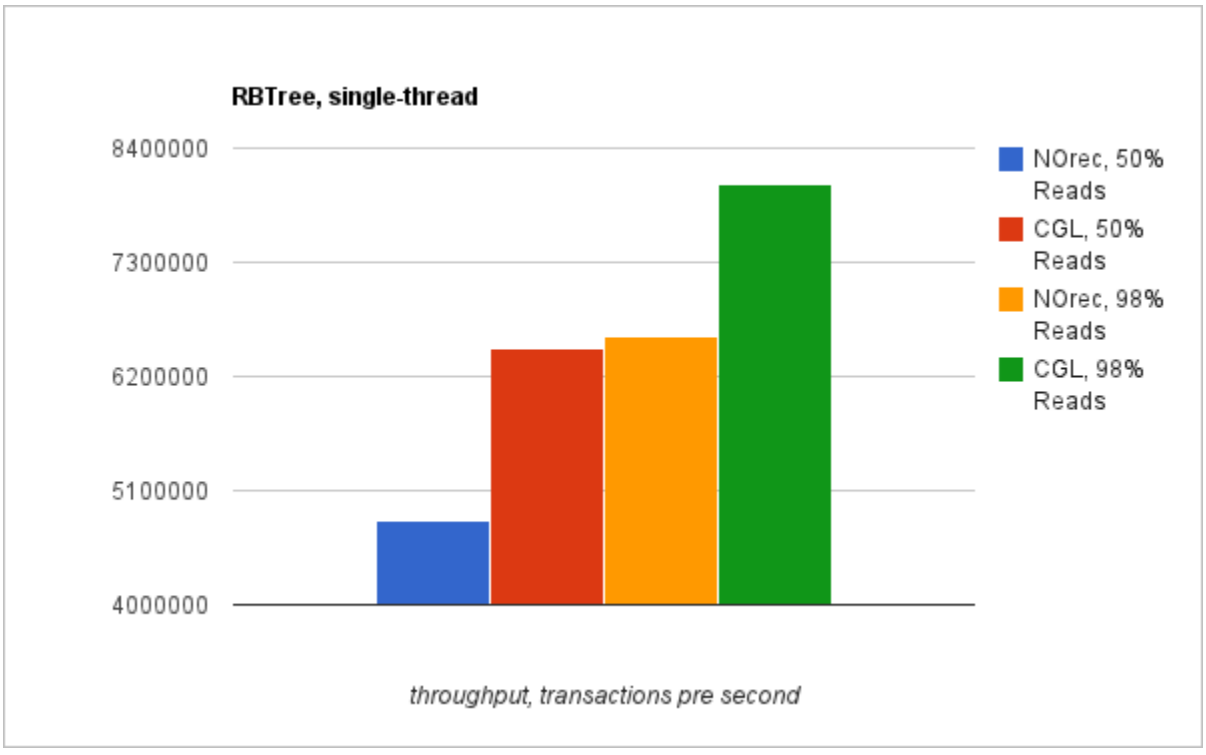


Figure 1

We present transaction throughput results in Figure 1. From the figure we can see that in either workload, the CGL have more throughput than NOrec, that indicate write buffering and

logging for validation have clear overheads, but these are the price of writer concurrency and privatization safety.

## Scalability

We use the same RBTree benchmark as in the Overhead to assess scalability in the same benchmark configurations. We run the test from total of one threads to 20 threads to measures the scalability of the algorithms. Figure 2 summarizes these results.

Since CGL is just a global lock, so it does not scale and it can be confirmed from the figure. The NOrec perform very well for read-mostly workloads, it outperforms in both scalability and throughput than CGL. NOrec has a single-writer commit protocol, which clearly limits its scalability in workloads with many writers, this can be confirmed from figure in different workloads, the 98% reads's throughput is almost 40 times than 50% reads's when the threads reach total of 16. Also from the blue line in the figure, we noticed that it shows very little changes in workloads with many writes, it is also because the single-write commit protocol. In the end, after the running threads more than total hardware threads, 16 in this case, we observed that the NOrec reach its limit and show a little throughput reduced after that.

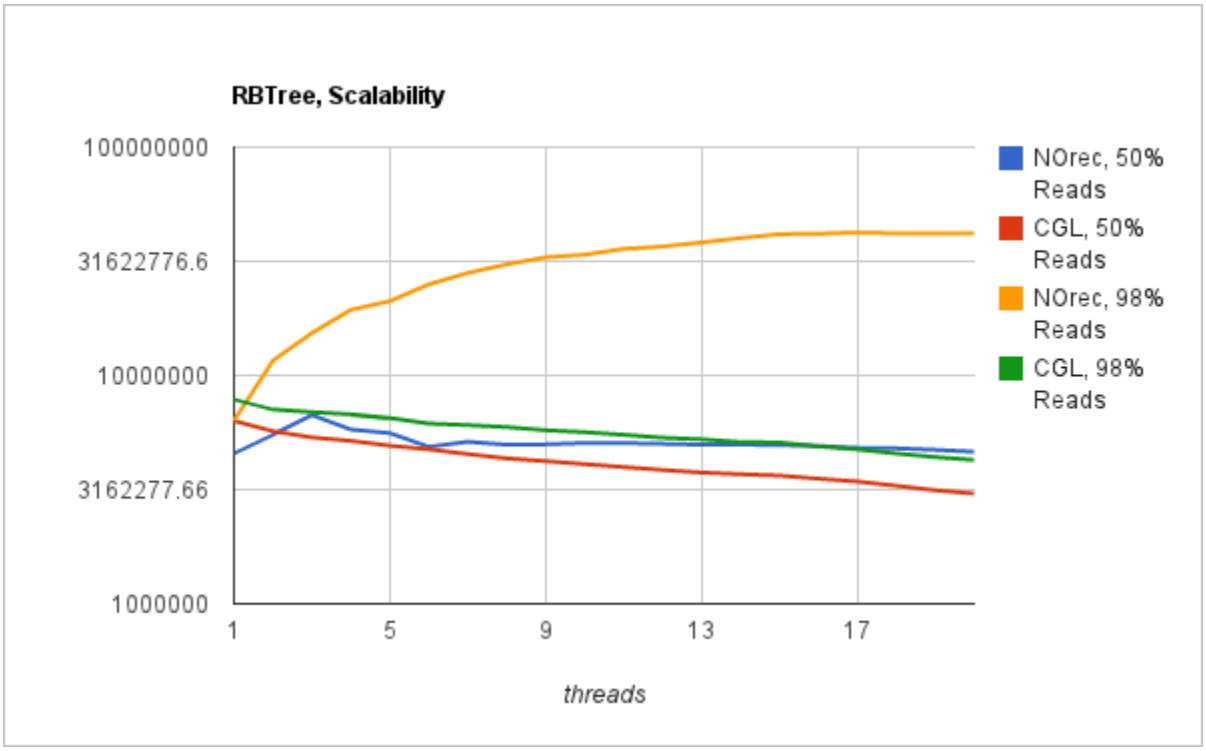

Figure 2

## Programmer's Perspective

In addition to their performance benefits, STM greatly simplifies conceptual understanding of multithreaded programs and helps make programs more maintainable by working in harmony with existing high-level abstractions such as objects and modules [7]. Compared to lock primitives, the concept of a memory transaction is much simpler, because each transaction can be viewed in isolation as a single-threaded computation. Deadlock and livelock are either

prevented entirely or handled by an external transaction manager; the programmer need hardly worry about it. Priority inversion can still be an issue, but high-priority transactions can abort conflicting lower priority transactions that have not already committed.

# Open Problems

## TM Is Not Panacea

Transactional memory cannot avoid all concurrency bugs automatically without the awareness of programmers. It is still regrettably easy to write incorrect code. As an example, the following code will even never stop, while it works well without TM.

```
bool flagA = false; bool flagB = false;
```

```
Thread 1:
atomic {
  while (!flagA);
  flagB = true;
}
```

```
Thread 2:
atomic {
  flagA = true;
  while (!flagB);
}
```

## Weak Isolation and Strong Isolation

So far it is not clear how transactions interact with access to shared data outside transactions. With weak isolation, a conflicting memory reference executed outside of a transaction may not follow the protocols of the TM system. Consequently, the reference may return an inconsistent value or disrupt the correct execution of the transaction. The exact consequences are implementation-specific. As a result, it is a programmer's responsibility to make sure no undesirable data race occurs. RSTM only supports weak isolation.

In contrast, strong isolation automatically converts all operations outside an atomic block into individual transactional operations, thus replicating the database model in which all accesses to shared state execute in transactions.

## I/O and System Calls

The need to abort failed transactions also places limitations on the behavior of transactions: they cannot perform any operation that cannot be undone, including most I/O and syscalls. In RSTM, we use transaction declaration explicitly, so I/O and system calls cannot be transactional.

## Memory management

Since the OS memory allocation and deallocation do not support transaction and they cannot be avoided in reality, we need to take care of this specific problem. To deal with it, a garbage collecting allocator may be used, or applications can use a transaction-aware allocator so that it can handle the memory allocation gently when a transaction fails.

# Conclusion

As a new programming model, STM relieve programmers from difficult and error-prone parallel programming by providing a clear and natural way to express their intention. STM has recently been the focus of intense research and support for practical implementations is growing. Due to the conceptual advantage of TM, we believe that hardware TM will eventually be seen as desirable, even necessary, in many-core machines. In the meantime, there are hundreds of millions of multicore machines already in the field. For the sake of backward compatibility, emerging TM-based programming models will need to be implemented in software on these machines. It also appears likely that software TM (STM) will be needed as a fall-back on future machines when hardware resources are exhausted.

# Reference


1. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 2008.
2. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*. 2008.
3. D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2009.
4. J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2007.
5. L. Dalessandro, M. Spear, and M. Scott. Norec: Streamlining STM by Abolishing Ownership Records. In *Proc. of the ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 67–78, Jan 2010.
6. M. F. Spear, A. Shriraman, L. Dalessandro, and M. L. Scott. Transactional Mutex Locks. *SIGPLAN Workshop on Transactional Computing*, Feb. 2009.
7. Software transactional memory, http://en.wikipedia.org/wiki/Software_transactional_memory, April 2012