Extensible Transactional Memory Testbed[☆]

Derin Harmanci[a,1], Vincent Gramoli[a,b], Pascal Felber[a], Christof Fetzer[c]

[a]*University of Neuchâtel, Rue Emile-Argand 11, CH-2009 Neuchâtel, Switzerland*
[b]*EPFL, Station 14, CH-1015 Lausanne, Switzerland*
[c]*Dresden University of Technology, D-01062 Dresden, Germany*

Abstract

Transactional Memory (TM) is a promising abstraction as it hides all synchronization complexities from the programmers of concurrent applications. More particularly the TM paradigm operated a complexity shift from the application programming to the TM programming. Therefore, expert programmers have now started to look for the ideal TM that will bring, once-for-all, performance to all concurrent applications. Researchers have recently identified numerous issues TMs may suffer from. Surprisingly, no TMs have ever been tested in these scenarios. In this paper, we present the first to date TM testbed. We propose a framework, TMunit, that provides a domain specific language to write rapidly TM workloads so that our test-suite is easily extensible. Our reproducible semantic tests indicate through reproducible counter-examples that existing TMs do not satisfy recent consistency criteria. Our performance tests identify workloads where well-known TMs perform differently. Finally, additional tests indicate some workloads preventing contention managers from progressing.

Key words: Transactional memory, semantics, performance.

1. Introduction

Transactional Memory (TM) is a programming language abstraction that simplifies concurrent programming by hiding complex synchronization primitives. Programmers can make a program thread-safe by simply labeling regions of sequential code as transactions: either all the transaction changes take effect, the transaction *commits*, or none of them take effect, the transaction *aborts*.

The drawback of the TM paradigm is the induced complexity inherent to the development of such an abstraction. Democratizing concurrent programming initiated a complexity shift from the application level to the TM level itself, and the new challenge is to develop a TM that would be reasonably efficient under a various set of workloads.

In this paper, we address this issue by testing a series of TMs. More precisely, we test their semantics to make sure they fulfill the expectations of application programmers and we test their performance to make sure their efficiency remains acceptable whatever the contention can be.

For the past few years, researchers from industry and academia have devoted much effort to study transactional memory semantics and to highlight important open questions. One question asked in [2] is crucial for the compliance of new transactional code with non-transactional legacy code: How should a transaction behave in presence of concurrent non-transactional accesses (in a weak-atomicity model)? Another question is to consider whether it is a waste of time to check that a transaction must abort [17]: Should a transaction abort even though its commit would not violate consistency? A final question concerns the TM guarantees when the memory model is not defined [41]: What result could we expect from a TM implementation when the memory model of the application language relaxes the program order of the code running on each thread?

These questions indicate the essential role of specific interleavings of conflicting operations in transactions. Various examples have been described as code fragments that may expose anomalies when transactional operations are executed in a certain order [2, 17, 23, 36, 41, 49, 52] but no tests of real TMs have been reported.

Testing concurrent programs in a reproducible manner is not an easy problem [13, 42, 55]. One might think of recording the events of the execution at run-time to replay them later on. Unfortunately such recording may directly affect the way events are interleaved. Exhaustive testing of concurrent executions is also tedious, when practical, as the number of executions to test grows exponentially with the number of events each thread executes. Conversely, the amount of possible transactional operation executions is much more reasonable, which makes them testable. We believe that developing a TM is the task of skilled programmers that are used to avoiding lower-level concurrent

[☆]Part of this work has already been presented at the non-archiving ACM SIGPLAN Workshop on Transactional Computing [25].

[1]Corresponding author. Address: Rue Emile-Argand 11, B-119, CH-2009 Neuchâtel, Switzerland, Phone: +41 32 718 2734, Fax: +41 32 718 2701, E-mail address: derin.harmanci@unine.ch.

programming issues. Our goal is thus to evaluate TMs on reproducible transaction operation executions to address existing open issues.

Initially: x=0, y=0

Thread 1	Thread 2
atomic {	
x=1;	
	y=x;
x=2;	
}	

```
Definitions:   x = 0; y = 0;
Transactions:  T := W(x,1), @L, W(x,2);
Schedules:     S := T@L, { y=x }, T;
Invariants:    [y != 1];
```

Figure 1: Dirty read test: a simple pathological scenario that may lead to a dirty read ($y = 1$) on the left-hand side, and the corresponding specification to test if the TM avoids the dirty read on the right-hand side. Note that $y = x$ is executed outside transactions and we define a label $@L$ in transaction T to specify the interleaving of operations in the schedule S.

Consider, for instance, the well-known problem of dirty reads. On the left-hand side of Figure 1, the first thread executes a transaction (represented as an `atomic` block) that updates the same location x twice while the second thread concurrently reads location x. Since the transaction should appear as if it were executed atomically, it should not be possible to have $y = 1$. This would correspond to a dirty read of location x by the second thread. Observe that, when both threads execute in parallel, some interleavings of operations may never produce dirty reads. To test appropriately that a given TM avoids this problem, we propose, on the right-hand side of Figure 1, the specification of a dirty read unit test. The interleaving of the transactions is defined in a schedule that enforces the read operation of x by the second thread to occur between the two atomic write operations of the first thread. The invariant checks whether a dirty read occurs in this specific schedule. Such a test language specification is of crucial importance for testing TM behaviors in particular situations, notably in scenarios with concurrent transactional and non-transactional accesses.

In this paper, we present the first to date TM extensible testbed. We proceed by first describing our testing framework TMunit dedicated to ease the specification of TM workloads. The results indicate unexpected caveats in recent TM implementations. For instance, we are the first to show that WSTM violates opacity and virtual-world synchrony and we provide a counter-example as a unit-test for TM. Finally, our test-suite is easily extensible for future use by the TM community as it builds upon a novel language for specifying TM workloads.

1.1. Contributions.

We propose the first to date TM testbed. Some of our tests have recently been proposed in the literature as potential problematic workloads on which TM would behave unexpectedly. Unfortunately none of these problems have been experimented on real TM systems as far as we know.

Our testbed relies on the use of a novel framework, TMunit, which we have especially designed to provide a domain-specific language dedicated to TM workloads. TMunit allows users to rapidly specify benchmarks to evaluate and compare TMs, as well as unit tests to validate the behavior of a specific implementation. TMunit runs a given TM on some, possibly randomized, workload and records the performance statistics. The specified benchmarks can be executed by dynamically interpreting the workload specification and mapping transactional accesses to an underlying TM, or for best performance, by generating a corresponding program to be compiled into a standalone application.

We applied our testbed to six state-of-the-art TM libraries that are $\mathcal{E}$-STM, RSTM, SwissTM, TinySTM, TL2 and WSTM (while we use the term "TM" for generality, we only consider *software* transactional memories, STMs, in the remainder of the paper). Our main results (i) demonstrate which TMs fail on pathological scenarios because of consistency violation, presence of non-transactional code or over-conservative semantics, (ii) indicate that efficient TMs may have very poor performance in very specific circumstances while alternative TMs may perform better, (iii) show schedules in which WSTM violates opacity and virtual-world synchrony [30] in case no additional validation is performed by the programmer, and (iv) illustrate a livelock execution due to the use of Passive contention manager [53].

Besides these results, we believe that TMunit will be helpful not only for TM developers to validate and improve performance of their TM, but also for TM users to choose the most efficient TM support for their specific application workload.

1.2. Availability

Our testbed including TMunit and the necessary files to test new TM programs are available at `http://www.tmware.org/tmunit`.

1.3. Roadmap

In Section 2, we present TMunit, the new testing framework our results are based on. In Section 3, we briefly introduce the state-of-the-art TM libraries we test in the paper. Sections 4 and 5 test, respectively, the semantics and the performance of the TMs while Section 6 illustrates the performance of contention managers. Section 7 lists the work related to the testing of concurrent programs and Section 8 concludes the paper.

2. TMunit

Evaluating TMs requires to test their behavior not only in response to minimal workloads (by unit testing) but also in response to more complex workloads (by performance testing). In this section, we describe TMunit, the testing framework we have developed to rapidly write performance and semantics tests for TMs.

2.1. Overview

Here, we describe the main components of TMunit their interactions being depicted in Figure 2. TMunit executes a synthetic workload written in a domain-specific language, on a dedicated TM, and records performance statistics and test results. To this end, TMunit uses a parser to transform the workload into an executable. This parser is written using the `lex` and `bison` tools. Depending on the choice of the user, it can either output an *interpreted* automaton, to execute the workload dynamically, or a *generated* automaton, to reduce the runtime overheads.

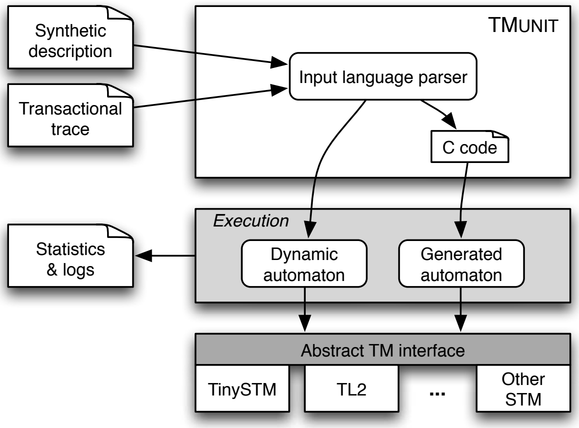


Figure 2: Architectural overview of TMunit.

2.1.1. Automata

The interpreted automaton is a data structure representing the workload. Typically, this data structure comprises a set of linked lists, each representing a transaction whose nodes are the operations to execute. This data structure allows loops and conditional executions.

The generated automaton is the translation of the configuration file into source files in a specific programming language.[2] These source files can then be compiled and executed as a stand-alone application.

While the interpreted automaton can be used to interpret and execute the configuration file in a one-step process, the generated automaton executes with negligible overheads, as shown later in Section 5. In contrast, the interpreted automaton is more convenient for execution of simple unit tests, e.g., when testing TM semantics.

2.1.2. Execution

The automata can be executed in two different modes: the schedule mode and the parallel mode. In the schedule mode, the execution corresponds to the sequence of transactional operations described in the schedule definition of the given workload. Each transaction is executed in a separate thread and only one thread is active at a time. This mode is mostly convenient for unit tests (see Section 2.3). TMunit can also automatically generate schedules and execute them. In such a case, TMunit will generate schedules that correspond to all possible interleavings of the transactional operations (where each transaction runs on a separate thread). In the parallel mode, the execution is performed according to a thread specification described in the given workload. In this execution mode threads execute concurrently.

2.1.3. Abstract TM interface

Each automaton communicates with the underlying TM using a standard TM interface to initialize the transactional memory and thread data structures and to delimit transactions, e.g., `start` and `commit`, but also to execute transactional operations, e.g., `read` and `write`. TMunit has generic calls for these operations and the collection of these generic calls form its abstract TM interface. To plug a new TM to TMunit a user simply needs to map a transactional operation of this TM to its corresponding generic TMunit call (using the header file), and to recompile TMunit.

Although this paper focuses on the results obtained with software transactional memory, the interface is generic enough to support hardware transactional memory (HTM) as well. TMunit has been successfully adapted to be used with AMD's Advanced Synchronization Facility (ASF) [1] instruction set, which provides HTM support. The adaptation relied essentially on replacing software barriers with hardware barriers. TMunit has been used to test and identify bugs in ASF.

2.2. Simplicity and expressiveness

The language has been designed to be simple enough to specify transactions and schedules in an abstract way as usually found in academic papers [2, 17, 23, 36, 41, 49, 52] and expressive enough to reproduce classical transactional benchmarks using multiple data structure accesses.

```
T1 := R(x), R(y), W(x), W(y);                              // R = read, W= write
T2 := R(x,_a), R(y,_b), W(x,_a−10), W(y,_b+10);          // _a, _b = thread locals
```

Listing 1: Two sample transactions.

Listing 1 illustrates how simple it is to specify basic workloads. The first transaction, T_1, reads two memory locations before updating them (note that transaction beginning and commit are implicit). Memory locations are designated by symbolic addresses that will be mapped to shared memory by TMunit. Here, we are not interested in the value read or written, i.e., we are only interested in possible conflicts. In contrast, T_2 stores the values read in local variables and writes updated values to shared memory, similar to a transfer between bank accounts. One can specify far more sophisticated behavior in transactions, as will be discussed next.

A workload (unit test or performance test) is written as a configuration file divided into six sections:

[2]Currently, the C language is supported for generated automata.

1. The *properties* section presents the execution settings and parameters.
2. The *definitions* section specifies the variables and constants.
3. The *transactions* section defines the operations that compose each transaction, using a simple but sufficiently powerful language.
4. The *threads* section specifies each thread as a transaction pattern.
5. The *schedules* section describes specific executions with a pre-determined interleaving of operations.
6. The *invariants* section specifies assertions that must be valid at each step of a schedule.

2.3. Unit test specification

Here, we consider a specific kind of tests especially suited for TMs: they represent a deterministic scenario of a parallel execution. As motivated in the introduction, there is a crucial need for unit testing TMs to outline problems due to certain interleavings of conflicting operations. Note that unit tests in TMUNIT can include interleavings with non-transactional accesses, which allows testing strong atomicity support of TMs (as in Figure 1). Listing 2 illustrates our domain-specific language on the *zombie transactions* example of [2] (we have actually reproduced the equivalent variant of the zombie transactions example as sent by Dan Grossman on the *tm-languages* mailing list on July 1, 2008). The write to z by T_2 on Line 6 is dead code under single-lock semantics and should not happen. However, some TM implementations with eager update might perform the write and undo it later, causing the assertion to fail. Such a unit test can help determining whether the TM provides single global lock atomicity.

```
 1  Definitions:                                          // variables and constants
 2    y = 0; x = 0; z = 0;                          // shared variables, initially all 0
 3
 4  Transactions:                                    // specification of transactions
 5    T1 := W(x,1), @L1, W(y,1);                          // W = write, @L1 = label
 6    T2 := {? [ R(x) != R(y) ] : W(z,1) };        // R = read, {?:} = if statement
 7
 8  Schedules:                                          // specification of schedules
 9    S := T1@L1, T2, T1;                   // execute T1 until L1, then T2, finish T1
10
11  Invariants:                                             // invariants to fulfill
12    [z != 1];                                            // unprotected read of z
```

Listing 2: Unit test for zombie transactions [2].

2.3.1. Operations and transactions.

We assume a single address space of bounded size. Threads can only communicate by writing to, and reading from, the shared address space. We denote reads by `R` and writes by `W`. These two operations can only be applied to shared memory variables. Variables are defined in the *definitions* section and are either integers (of the size of a memory word) or arrays of integers. One can also use thread-local variables. Their name must start by an underscore symbol '_' and is scoped at the level of the transaction (they can be referred to as `<tx-name>:<var>` to avoid ambiguity). Variable names with only capital letters are considered as constants and their value cannot be modified. A read operation accesses a shared variable, or an entry in a shared array. The read operation returns the content of the shared variable as provided by the underlying TM. The result can optionally be recorded in a thread-local variable. We denote this by `R(<sh-var>)` or `R(<sh-var>, <loc-var>)`. Similarly, a write operation accesses a shared variable `W(<sh-var>)` to write a value that can be optionally specified `W(<sh-var>, <val>)`. We refer to shared variable accesses via read and write operations as *protected accesses*, and to direct shared variable accesses (e.g., `x = 0`) as *unprotected accesses*. TMUNIT supports arithmetic expressions involving numbers, variables, random values, arithmetic operators, and parentheses that yield an integer value.

Each transaction is given a unique name and represents a finite sequence of operations, delimited by commas, implicitly started by a "begin" statement and ended by a "commit". It is possible to explicitly abort a transaction by using notation `A` to implement sophisticated test scenarios. Inside a transaction and between operations, labels can be specified by `@<label>` and local variables can be assigned values. In Listing 2, T_1 contains two operations (Line 5) while T_2 contains one operation and an *if* statement with two operations (Line 6). Label `@L1` is used in T_1's definition as a marker for specifying the schedule as explained below.

2.3.2. Schedules and assertions.

Schedules specify a pre-defined interleaving of the transactional operations for testing or debugging a TM. If no schedules are predefined, all different schedules of transactional operations will be automatically generated and tested, which may take time. If schedules are specified they are defined in the *schedules* section and they specify the execution order of the transactions operations using `<tx-name>@<label>` to indicate that `<tx-name>` executes alone until label `@<label>`. If no label is specified, the transaction executes until the end. Note that, during the execution of a schedule, each transaction executes in its own thread, but only one thread is active at each step of the schedule. This is a design choice in order to provide repeatable schedules and unit tests. Multiple schedules can be specified but only one will be executed at a time; this schedule can be specified using command-line parameters.

The scheduler acts as a sequential process executing transactional operations in turn. In schedule execution mode, TMUNIT creates a thread per each defined transaction and an additional *scheduler* thread. The scheduler thread performs the switching between threads thanks to the barriers that correspond to the labels in the transaction definitions. There are also two additional barriers; one is used only for the scheduler and the other is an initial common barrier for all the threads except the scheduler. The barriers are used to pass a token between the threads and at a given time there is only one thread that owns the token. Initially, the scheduler thread owns the token and it passes the token to the first scheduled transaction. The thread owning the token executes until it encounters the next barrier, which corresponds to a label in the configuration file, and then passes the token back to the scheduler thread. If the thread encounters no barriers/labels it executes until the end of the corresponding transaction before passing the token to the scheduler. Upon receipt, the scheduler thread passes the token to the next scheduled transaction's thread. This continues until the end of the defined schedule.

Invariants and assertions define tests that the execution must pass. Assertions are boolean expressions and can be specified in transactions or schedules as `[<bool-expr>]`. Invariants are assertions that are automatically evaluated at each step of a schedule. If an assertion evaluates to false or if an invariant is violated during the execution, then the test fails. The program prints an error message. The evaluation context of variables within boolean expressions used in assertions and invariants are as follows.

- Local variables are evaluated in the context of the transaction they are used in.
- Unprotected accesses, like `[z != 1]` (Line 12 of Listing 2), are evaluated by directly reading the memory location, i.e., without using the underlying TM.
- Protected accesses, like `[R(x) != 1]`, evaluate in a dedicated transaction that performs only the protected access.

An example schedule for the "zombie transactions" scenario [2] is presented at Line 9 of Listing 2. In this schedule, transaction T_1 executes up to label L_1, and then transaction T_2 runs before $T1$ resumes. As a result, this schedule forces T_2 to read x between the two writes of T_1. If T_2 reads a dirty value 1, it will update z (Line 6) and the invariant (Line 12) will be violated, leading to the failure of the test.

2.4. Performance test specification

Performance tests are generally longer than unit tests since they execute more complex specifications to measure the performance of a TM. More precisely, they use randomization and loops to test a large set of schedules. Note that these specifications do not define schedules, thus the threads run concurrently for performance tests. Here, we present additional language features on a slightly more complex example that corresponds to the complete specification of a widely used micro-benchmark: a sorted linked list.

The resulting TMunit benchmark is 28 lines of code written in our language (see Listing 3) while it was originally more than 1000 lines of code written in C (as it is available in the TinySTM distribution). This is mainly due to the unnecessary specification of threads and statistics management that are automatically handled by TMunit. This simple TMunit version will be experimentally compared to the original benchmark in Section 5.2.

```
 1 Properties:                                                        // global properties
 2   RandomSeed = 1;                                             // use random seed for RNG
 3   ReadOnlyHint = 1;                                        // tag read−only transactions
 4   Timeout = 10∗1000∗1000;                              // maximum test duration (us)
 5
 6 Definitions:                                                  // variables and constants
 7   SIZE = 4096;                                               // size of the list (constant)
 8   m[0 .. 2∗SIZE+1] = 0;                                 // memory range for list nodes
 9   NB = <1 .. SIZE>;                      // random value (constant) in range 1. . .SIZE
10   T2:_f = 0;                                // flag to alternate between adds and removes
11   T2:_v = 0;                                              // position of last added value
12
13 Transactions:                                         // specification of transactions
14   T1 := {# k = [0 .. NB−1] : R(m[2∗k]), R(m[2∗k+1]) },         // search element
15           R(m[2∗NB]) ;
16   T2 := {? [_f == 0] :                                            // add/remove element
17             {# k = [0 .. NB−1] : R(m[2∗k]), R(m[2∗k+1]) },          // add element
18             R(m[2∗NB]), W(m[2∗NB−1]),
19             { _f = 1, _v = NB }
20           |
21             {# k = [0 .. _v−1] : R(m[2∗k]), R(m[2∗k+1]) },       // remove element
22             R(m[2∗_v]), R(m[2∗_v+1]),
23             W(m[2∗_v−1]), W(m[2∗_v]), W(m[2∗_v+1]),
24             { _f = 0 }
25           } ;
26
27 Threads:                                                   // specification of threads
28   P1, P2 := < T1 : 80% | T2 : 20% >∗;
```

Listing 3: Complete specification of the sorted linked list micro-benchmark.

2.4.1. Randomness and loops.

To implement realistic performance tests, our language provides powerful constructs such as random executions and loops. Randomness is provided by special constructs `<min..max>` that evaluate to an integer value chosen uniformly at random between `min` and `max` (inclusive). This random expression notation can appear everywhere a number is expected. For instance in Listing 3, constant `NB` at Line 9 represents an arbitrary element of a linked list of size 4096. Note that "random constants" are evaluated once at the beginning of each transaction, i.e., they get a different, immutable value for every transaction execution.

Transactions may include loops that repeat a predetermined number of times and loops that execute until a conditions becomes true. The former type of loop is illustrated in Listing 3 Line 14, where transaction T_1 repeatedly reads addresses representing nodes in the linked list (each node has two data items: a value (read using `R(m[2*k])`) and a pointer to the next node (read using `R(m[2*k+1])`). This transaction mimics the search for a random element in a linked list, with a number of iterations determined by the random constant `NB`. The last operation correspond to the read of the searched value (or the first larger value in case it is not found).

Conditional execution is another important mechanism to specify realistic workloads. Our language supports a generalized form of if-then-else statement. Conditional expressions may depend on the state of variables and constants. For instance, in Listing 3, transaction T_2 uses a flag $\_f$ to alternatively add or remove an element. If the flag is 0 then a new element is added (Lines 17–19); otherwise the last inserted element is removed (Lines 21–24). This approach is used by linked list micro-benchmarks to maintain the size of the list almost constant during the whole experiment. Note that the reason there is a single write

(`W(m[2*NB-1])` at Line 18) upon node insertion is that the new node is not shared until commit time; in contrast there are three writes upon removal (the writes at Line 23) because one needs to detect concurrent accesses to the removed node, which can be achieved by overwriting it. This specification closely mimics the behavior of a custom linked list micro-benchmark with the notable exception of the placement of the node data in memory (deterministic vs. unpredictable placement); yet, as we shall see in Section 5, this difference does not affect the results of performance tests.

2.4.2. Threads and transaction patterns.

The *threads* section specifies the combination of transactions that will execute in the context of each thread. Unlike transactions, threads may have infinite length and are defined as patterns using a syntax close to regular expressions. Each thread that executes at runtime must be defined. By default, the benchmark will execute one instance of each thread but command-line parameters can be used to indicate which threads to start and their number of instances (threads are referred to by their name). Multiple thread names can share the same specification. A thread definition may include repetitions (fixed, random, or unbounded), execution of one out of several transactions chosen at random with predetermined probabilities, sequences and grouping of transactions. As an example, Listing 3 presents two threads P_1 and P_2 that both execute transactions T_1 with probability 80% and transaction T_2 with probability 20% in an endless loop. Such experiments are interrupted after a specified timeout or with a signal.

3. TM Libraries

In this section, we introduce the six state-of-the-art TM libraries that we evaluate.

WSTM [26] is an early TM implementation written in C that associates version numbers to memory locations. WSTM uses an ownership table that maps memory locations to ownership records in order to track the rights of transactions to update locations. A hashing function is used to ensure that the size of the ownership records does not depend on the number of memory locations, hence the size remains fixed and two different addresses may map to the same ownership record. WSTM has been especially designed to ensure that a transaction can commit only if none of its ownerships protect a location modified by a non-committed transaction. In contrast, WSTM allows a transaction to access a location that has been modified by a transaction that did not commit yet. This implementation choice is mainly due to performance concern, however, WSTM provides the programmer with an explicit validation in case the programmer wants to make sure that a transaction accesses only committed values. This validation is not required and alternative sandboxing technique can be combined with WSTM to avoid infinite loops or division-by-zero errors.

TL2 [11] is a time-based TM implementation written in C where transactions log their write operations to redo them at commit-time using a two-phase-locking strategy. A transaction reading an address either returns the value logged in memory (if it already wrote it), or returns the value present in the shared memory (if it has never written it). All transactions read and/or increment a global counter to efficiently detect at commit time whether a read value has been modified. If the value has been modified the transaction has to abort, otherwise it can commit. Different global counter managements have been proposed to enhance scalability by minimizing the contention on this global counter, e.g., gv5 and gv6 [34]. Read operations are invisible so that a transaction that reads an address does not prevent other concurrent transactions from modifying this address. As a drawback, TL2 algorithm requires a transaction to execute completely before detecting that one of its first read operations has been invalidated by another transaction.

TinySTM [14] is another time-based TM implementation written in C. This algorithm builds upon LSA [48] but uses a single version per address for the sake of efficiency. More precisely, TinySTM chooses a timestamp interval for the serialization point of each transaction and may try to extend it during the transaction execution, as opposed to TL2 that never extends it. For instance, if a transaction accesses an address with a version number above its timestamp interval, it tries to extend its interval by making sure that none of its read addresses have been updated by a concurrent transaction. It aborts only if the extension is impossible whereas TL2 aborts in any case. TinySTM can be parameterized to either defer the write to commit-time as in TL2, we refer to this version of TinySTM as *write-back* and denoted by WB, or to execute immediately the writes in-memory, we refer to this version of TinySTM as *write-through* and denoted by WT. TinySTM can be parameterized to use one among two different locking strategies. The first locking strategy, is similar to TL2 in the sense that transactions lock all addresses to write at commit-time, this locking strategy is called *commit-time locking* and is denoted by CTL. The second locking strategy is different as transactions lock the addresses as soon as the transaction executes a write, this strategy is called *encounter-time locking* and is denoted by ETL.

RSTM [10] is a transactional memory library that can be configured in various ways and that was originally object-based. A word-based version of RSTM that resembles the TL2 algorithm has been proposed. We use both object-based and word-based versions of RSTM. On the one hand, the object-based version adopts a lazy acquirement strategy hence the write operations and their corresponding lock acquirement are logged during the transaction execution and deferred to commit time. In addition, the reads are invisible in the sense that no transactions can detect that another transaction has read a value. Finally, the object-based version runs by default a priority-based contention manager named Polka [53]. On the other hand, the default word-based version of RSTM is timestamp-based and also adopts a lazy versioning and lazy acquirement strategy so that write operations are deferred to commit time similarly to TL2 and the write-back commit-time locking version of TinySTM. Again, there are multiple ways to configure RSTM and we execute it in its default configuration mode except in Section 6 where we use its eager acquirement strategy to test the contention managers.

SwissTM [12] is a C++ implementation similar to TinySTM in that it uses a global counter and an interval extension. SwissTM combines the encounter-time with the commit-time locking strategies because it detects write-write conflicts eagerly and read-write conflicts lazily. Eager write-write detection is interesting as it avoids wasting efforts in executing the whole transaction before detecting that a roll-back is necessary. Lazy read-write conflict detection is interesting as it avoids the most common type of unnecessary aborts where a read address is overwritten by a concurrent transaction before commit-time [17]. In contrast with other STMs, SwissTM also comprises an adaptive contention manager strategy that uses the passive contention manager [53] ideal for small transactions and the greedy contention manager better suited for long transactions.

$\mathcal{E}$-STM [15] provides elastic transactions, a recently proposed transactional model that ensures atomicity only at the application level to enhance concurrency. Besides providing also regular transactions, in elastic ones it ensures that only write operations and couples of consecutive operations executed by the same transaction are atomic. Its novelty lies in the combination of these elastic transactions whose size is determined during execution, with regular transactions whose implementation is built upon TinySTM-ETL. More precisely, an elastic transaction ensures that all its write operations plus the read that precede them (in case there is one) looks like a regular transaction and it ensures that all couples of consecutive operations look like regular transactions as well. For instance, if an elastic transaction comprises three consecutive read operations on locations x, y, z, then the reads of x and y form a regular transaction and the reads of y and z form another regular transaction, but the three reads do not form a regular transaction.

4. Testing TM Semantics

As already mentioned, many issues related to the semantics of TMs have been identified in the literature. Those range from "unnecessary abort" [17] to "publication" [41] and are usually expressed as an interleaving of operations executed by few threads. They may lead to unexpected results when executed on a given TM. In this section, we present several semantic tests. Their results are summarized in Table 1.

4.1. Semantics

We first enumerate a series of consistency criteria that specify the behavior a transactional system should adopt. The goal here is not to give formal definitions of consistency criteria but rather to recall the reader with the general meaning of these criteria; for further details please refer to the associated cited papers.

- *Serializability* is a consistency criteria originally defined in [45] that has been extensively used to characterize transactional database systems. For the sake of simplicity, we consider in the remainder only read-/write objects and we assume that all executions respect the sequential specification of these objects meaning that a read must return the last written value or the default one in case no such write exists. Serializability requires that any execution of the system must appear equivalent to an execution where all its transactions would have executed sequentially.

- *Linearizability* was defined for non-transactional operations [29]. The application of linearizability to transactional systems requires that all executions must be equivalent to an execution where transactions would be executed sequentially and where non-concurrent transactions[3] must be in the same order. Linearizability is a strictly stronger consistency criterion than serializabilty.

- *Opacity* aims at preventing transactions from accessing inconsistent states that would result in division-by-zero errors or infinite loops [23]. Opacity requires that *all committed or aborted transactions* appear as if they were executed atomically in an order satisfying the real-time order and that all transactions including aborted ones access only consistent system states at any time.

- *Elastic-opacity* applies to high level operations that require much weaker guarantees than the previous criteria propose [15]. Typically, elastic-opacity assumes a model with two types of transactions, elastic and normal, and requires that if we cut elastic transactions into smaller sub-transactions, then we obtain an execution composed of normal transactions and sub-transactions that is opaque. Elastic-opacity guarantees operation atomicity at the application level, its goal is to enhance concurrency by relaxing the unnecessary atomicity provided at read/write level. Elastic-opacity is a strictly weaker consistency criterion than opacity.

- *Single global-lock atomicity (SGLA)* is a criterion simple to reason with as it describes the semantics of

[3]Two transactions are non-concurrent if there is no common point in time where the two transactions have started and none of them have terminated yet.

```
Definitions:
x=0; y=0; _t=0;
Transactions:
T1 := W(x,1);
T2 := R(x), @L, R(y,_t);
T3 := W(x,2), W(y,2);
Schedules:
S := T1, T2@L, T3, T2;
Invariants:
[_t!=2];
```

```
...
    [Th3:T3] W(y)
    [Th3:T3] W(y,2)
    [Th3:T3] Try C
    [Th3:T3] C
[Th2:T2] R(y)
[Th2:T2] R(y,2)
Invariant '[_t!=2]' failed.
```

```
Definitions:
x=0; y=0;
Transactions:
T1 := R(x), @L, R(y);
T2 := W(x);
T3 := W(y);
Schedules:
S := T1@L, T2, T3, T1;
Invariants:
[No-abort];
```

```
...
        [Th3:T3] S
        [Th3:T3] W(y)
        [Th3:T3] W(y,23264)
        [Th3:T3] Try C
        [Th3:T3] C
[Th1:T1] R(y)
[Th1:T1] A
Invariant NO_ABORT fails.
```

```
Definitions:
x=0; y=0; z=0; t=0;
Transactions:
T1 := W(x), W(y);
T2 := W(z), W(t);
T3 := R(z), @L2, W(y);
TL := R(x), R(y), @L1, R(z), W(t);
Schedules:
S := TL@L1, T3@L2, T1, T2, T3, TL;
Invariants:
[No-abort] ;
```

```
...
[Th2:T2] W(t)
[Th2:T2] W(t,27225)
[Th2:T2] Try C
[Th2:T2] C
    [Th3:T3] W(y)
    [Th3:T3] W(y,23264)
    [Th3:T3] Try C
    [Th3:T3] A
Invariant NO_ABORT fails.
```

Figure 3: **(Left)** The opacity test for which WSTM fails while other TMs succeed. WSTM trace appears below. **(Middle)** The linearizability violation test for which all TMs pass. The tests terminates with an abort failure which indicates linearizability is not violated. The the trace for TL2 appears below. **(Right)** The serializability violation test that all TMs pass. The tests terminates with an abort failure indicating that serializability is not violated. The trace for TL2 appears below.

transactions to be as if all transactions access a single global lock during its whole execution [33].

- *Virtual world consistency* requires that *all committed transactions* appear as if they were executed atomically in an order satisfying the order of non-concurrent transactions and that all transactions including aborted ones access only consistent system states at any time. Hence, this criterion is weaker than opacity as it allows the set of all transactions (aborting and committing) to not appear as executed atomically, as long as the subset of committed transactions does so [30].

In the upcoming parts, we could not run all tests from the articles cited above because some tests (e.g., granular inconsistent reads of [41]) depend on the memory model used. As TMunit is independent from any memory model, those tests would require the user to indicate directly in the configuration file, which optimizations/constraints its targeted memory model allows/ensures. Although it could be a very interesting future work, comparing memory models remains out of the scope of this paper.

4.2. Safety tests

The safety tests are interesting to better understand the consistency criterion ensured by TMs. Some TMs ensure opacity [23], some other ensure serializability. As pointed out in [17], however, some serializable TMs may also be opaque.

Here, we chose four safety tests from the literature. The *opacity* test (1) presented in Figure 3 (left) comes from [23] (Fig. 1) and was used to illustrate the difference between strong atomicity and opacity requirements. The *linearizability* test (2) in Figure 3 (middle) and taken from [17] (Fig. 2), aims at showing that a serializable TM may not be linearizable. The *serializability* test (3) depicted in Figure 3 (right) and coming from [49] (Fig. 2) exhibits that a TM is not serializable. Finally, the *SGLA* test (4) illustrated in Figure 4 and given in [41] (Fig. 11) indicates how single-global-lock-atomicity (SGLA) [41] can be violated to allow more concurrency as the disjoint-lock atomicity would allow.

```
Definitions:
 x =0; y =0; z=0;
 t1=0; t2=0;
_u =0;
Transactions:
 T1 := W(z,1);
 T2 := R(x,_u), W(t1,_u), @L,
      W(y,1);
Schedules:
 S:= T2@L, {x = 1}, T1,{t2 = y},
     T2, [!((t1==0) && (t2==0))]
Invariants:
 [No-abort];
```

```
...
-------  x = 1  -------
[Th1:T1] S
[Th1:T1] W(z)
[Th1:T1] W(z,1)
[Th1:T1] Try C
[Th1:T1] C
-------  t2 = 0  -------
   [Th2:T2] W(y)
   [Th2:T2] W(y,1)
   [Th2:T2] Try C
   [Th2:T2] C
Assertion [!((t1==0)&&(t2==0))]
not satisfied.
```

Figure 4: The SGLA violation test for which all TMs fail. The tests terminates with the violation of the assertion in the schedule which indicates that SGLA is violated. Right-hand side shows the trace obtained from running TL2.

TL2, TinySTM, RSTM, and SwissTM pass all the critical tests we proposed except the SGLA test. Although this does not prove that they ensure the corresponding criterion, it simply shows that those TMs do not violate consistency in these specific scenarios. For the tests, $\mathcal{E}$-STM has been restricted to its elastic transaction implementation, and $\mathcal{E}$-STM fails to pass the linearizability and serializability tests. Since opacity is known to be strictly stronger than these two criteria we deduce that $\mathcal{E}$-STM violates also opacity. This was expected as the elastic transactions of $\mathcal{E}$-STM weaken intentionally the normal transactions for high level operations. Thus, testing the linearizability of an integer set rather than the read/write would reveal that $\mathcal{E}$-STM ensures linearizability, but at a higher level of semantics.

Unexpectedly, however, WSTM successfully passes the linearizability and serializability tests but not the opacity test which indicates that WSTM is not opaque. The detailed TMUNIT trace gave us some information about the reason of opacity violation: opacity as opposed to linearizability requires that no transaction (even though it aborts) can see the result of the modification of another concurrent transaction. As shown in the *interference* test of Figure 9 (top), WSTM allows this to happen. The design of WSTM intentionally separates the isolation from the transactional memory abstraction, however, the programmer can explicitly validate to avoid isolation issue or can assume sandboxing that will prevent isolation errors like division-by-zero. This observation raises the question whether isolation should be part of the transactional memory semantics, as recently suggested by opacity and virtual-world synchrony.

4.3. Unnecessary abort tests

It is often the case that TMs unnecessarily abort a transaction even though there is no conflict, e.g., because distinguishing between a likely conflict and a real one would be too expensive. When the cost of aborts is high, however, it is important that a transaction commits whenever there is no risk of violating safety. We refer to an *unnecessary abort* [17] as an abort that occurs at some transaction that could have committed safely. As unnecessary aborts may have a non-negligible impact on TM performance, it is important to identify them.

```
Definitions:
  x=0; y=0;
Transactions:
  T1 := R(x), @L, R(y);
  T2 := W(y,1);
Schedules:
  S := T1@L, T2, T1;
Invariants:
  [No−abort];
```

```
...
    [Th2:T2] S
    [Th2:T2] W(y)
    [Th2:T2] W(y,1)
    [Th2:T2] Try C
    [Th2:T2] C
[Th1:T1] R(y)
[Th1:T1] A
Invariant NO_ABORT fails.
```

```
Definitions:
x=0; y=0;
Transactions:
 T1 := W(x), @L;
 T2 := R(x);
Schedules:
 S := T1@L, T2, T1;
Invariants:
 [No−abort];
```

```
[Th1:T1] S
[Th1:T1] W(x)
[Th1:T1] W(x,14666)
[Th1:T1]:L
   [Th2:T2] S
   [Th2:T2] R(x)
   [Th2:T2] A
Invariant NO_ABORT fails.
```

Figure 5: **(Top)** Write-during-readonly test for which TL2 and word-based RSTM abort while other TMs commit (TMUNIT gives the same trace for TL2 and RSTM, represented on the right-hand side). **(Bottom)** Invisible-write test where TinySTM's ETL and WT variants fail to commit both transactions without aborting. Right-hand side shows the trace for TinySTM-ETL.

We test the six TM implementations on three unnecessary abort tests. The *write-during-readonly* test (5) is presented in Figure 5 (top). Word-based RSTM and TL2 fail this test while other TMs succeed. The reason for the success of TinySTM and SwissTM is because they both use the LSA validation extension mechanism while word-based RSTM and TL2 do not. $\mathcal{E}$-STM would have aborted only if T_2 would also write x in addition to y otherwise it considers that T_1 is serialized after T_2 and passes the test.

The *invisible-write* test (6) given in Figure 5 (bottom) comes from the [17] (Fig. 1) and checks whether unnecessary aborts may occur when the write operation is made visible. TinySTM-ETL and $\mathcal{E}$-STM fail the *invisible-write* test because they lock the to-be-written address eagerly as soon as a write occurs leading to a conflict later on.

```
Definitions:
  arr[0..1]=0;
Transactions:
  T1 := W(arr[0],1), @L;
  T2 := W(arr[1],2);
Schedules:
  S := T1@L, T2, T1;
Invariants:
  [No−abort];
```

```
[Th1:T1] S
[Th1:T1] W(arr[0])
[Th1:T1] W(arr[0],1)
[Th1:T1]:L
  [Th2:T2] S
  [Th2:T2] W(arr[1])
  [Th2:T2] A
Invariant NO_ABORT fails.
```

Figure 6: The false-sharing test for which SwissTM, $\mathcal{E}$-STM and word-based RSTM abort while the other TMs commit (the trace of SwissTM is given on the right-hand side).

The *false-sharing* test (7) described in Figure 6 accesses consecutive memory locations. Having a lock protect multiple consecutive addresses can have the undesirable consequence of producing false sharing, i.e., two threads accessing distinct memory locations can conflict because they share the same lock. As shown by this test, SwissTM, $\mathcal{E}$-STM and word-based RSTM abort, meaning they use a common lock on (at least) two consecutive addresses. Unlike other STMs, they are subject to false sharing.

```
Definitions:
 _SIZE = 3;
 m[0 .. _SIZE] = 0;
Transactions:
  T1 := R(m[0]),@L1,W(m[1]);
  T2 := W(m[0]);
  T3 := W(m[0]), W(m[2]);
  T4 := R(m[0]), R(m[1]),@L2,R(m[2]);
Schedules:
  S := T1@L1, T2, T4@L2, T3,T4,T1;
Invariants:
  [T1:No−abort];
```

Figure 7: The virtual-world test indicating whether an unnecessary abort happens in transaction T1. All TMs fail this test because they all abort T1 as if committing it would violate safety even though T4 has to abort).

The last unnecessary abort test is the *virtual-world* test (8). Virtual-world consistency is weaker than opacity as mentioned before because it allows aborting transactions to have a view of the system that is different from the view of other transactions (committed or aborted). The *virtual-world* test indicates an execution in which a transaction T_1 has to abort to ensure opacity while T_1 could commit without violating virtual-world consistency. Hence, when considering virtual-world consistency definition the abort of T_1 is unnecessary. We can see that all considered TMs fail this test as they unnecessarily abort T_1. We are not aware of any TM library that could ensure virtual-world consistency without ensuring opacity, and the efficiency of such an implementation remains an open question.

```
Definitions:
 data=42; ready=0;
 val=0 ; _tmp=0;
Transactions:
 T1 := W(ready, 1);
 T2 := R(data,_tmp), @L,
   {? [R(ready)==1] :
        W(val,_tmp)};
Schedules:
  S := T2@L, {data=1}, T1, T2;
Invariants:
  [val!=42];
  [No−abort];
```

```
...
-------  data = 1  -------
[Th1:T1] S
[Th1:T1] W(ready)
[Th1:T1] W(ready,1)
[Th1:T1] Try C
[Th1:T1] C
    [Th2:T2] W(val)
    [Th2:T2] W(val,42)
    [Th2:T2] Try C
    [Th2:T2] C
Invariant '[val!=42]' failed.
```

```
Definitions:
  x=0; y=0;
Transactions:
  T := W(x,1), @L, W(x,2);
Schedules:
  S := T@L, {y=x}, T;
Invariants:
  [y!=1];
```

```
[Th1:T] S
[Th1:T] W(x)
[Th1:T] W(x,1)
[Th1:T]:L
-------  y = 1  -------
Invariant '[y!=1]' failed.
```

Figure 8: **(Top)** The publication test where all TMs fail. Right–hand side shows the trace for SwissTM. **(Bottom)** The dirty-read test where only TinySTM's WT variant fails because both transactional and non-transactional writes immediately modify the memory. Right-hand side shows the trace for TinySTM-WT.

4.4. Isolation tests

We tested five critical scenarios among which three include non-transactional accesses. All considered TM implementations used here ensure *weak atomicity*: transactions appear as if they were atomic with respect to each other but not with respect to non-transactional accesses (we did not use the fences of RSTM that would counteract our schedules).

Specifically, we performed the following isolation tests. The *publication* test (9), as described in Figure 8 (top), outlines a possible difference between the TM semantics and the lock semantics of the Java memory model [41]. The *dirty-read* test (10) discussed in the introduction is specified in Figure 8 (bottom) and the *zombie transaction* (11) test has been specified in Listing 2. The *interference* test (12), described in Figure 9 (top), outlines a possible interference between two transactions. Finally, in Figure 9 (bottom), the *granularity* test (13) raises issues relying on the granularity of TM accesses when a coarse-granularity may have side-effect on locations that were not intentionally accessed.

As expected [41], all TMs fail the publication test (9). The reason is simply that none of the considered TMs can ensure atomicity when non-transactional code accesses shared data.

Only TinySTM-WT fails the dirty-read test (10), other TMs succeed. This is due to the write-through strategy with which updates are directly written to memory when encountering a write operation and can potentially be reverted upon abort. Note that the other TMs might also exhibit this problem if an unprotected read occurs while a transaction is committing, but this scenario is not specified by our schedule.

All TMs pass successfully the test of zombie-transaction (11). First, all the TMs that use the write-back strategy defer modification to commit time so that transaction T_2

```
Definitions:
  x=0; y=0; _v=0; _w=0;
Transactions:
  T1 := R(x,_v), @L, R(y,_w);
  T2 := W(x,1), W(y,1);
Schedules:
  S := T1@L, T2, T1, [_v==_w];
```

```
...
[Th1:T1] R(y)
[Th1:T1] R(y,1)
[Th1:T1] Try C
[Th1:T1] A
[Th1:T1] Terminates
Assertion [_v==_w] fails.
```

```
Definitions:
  arr[0..1]=0;
Transactions:
  T1 := W(arr[0],1), @L;
Schedules:
  S := T1@L,{arr[1]=1},
     T1, [ arr[1] == 1];
```

Figure 9: **(Top)** The interference test for which only WSTM, the trace of which is on the right side, fails. **(Bottom)** The granularity test.

reads value 0 for x. Second, TinySTM-WT aborts immediately T_2 when trying to read x; hence the read does not return and the invariant is not violated.

The role of *interference* test (12) given in Figure 9 (top) is to check whether a writing transaction can interfere with a concurrently reading transaction. For TL2, TinySTM, RSTM, SwissTM, and $\mathcal{E}$-STM interference was prohibited, meaning that the write could not interfere with an ongoing reading transaction, even if this ongoing transaction eventually aborts. Conversely, WSTM allows interference and makes a transaction read a concurrently written value before aborting. This interference is the reason why WSTM violates opacity and virtual-world consistency (this violation has been detected in Subsection 4.2), since opacity and virtual-world consistency both require that linearizability be satisfied and that transactions do not interfere with aborting transactions. As a result, if the user of WSTM is not aware of this subtle characteristics, then his/her application may behave unexpectedly: T_1 observing that x is different from y may provoke an infinite loop, a division-by-zero or an irrevocable external event, like missile firing [18].

A last isolation test we have specified is a *granularity* test (13) depicted in Figure 9 (bottom) similar to some of [52]. We obtained the following results: (i) no problem occurs for WSTM, TinySTM, TL2, SwissTM, word-based RSTM, and $\mathcal{E}$-STM; (ii) the problem occurs only for object-based RSTM. As expected, word-based STMs do not suffer from this issue as they do not buffer the whole array to roll it back. In contrast, the object based version of RSTM, which accesses the memory with the granularity of objects, fails the test. This is due to the fact that we consider the whole array as a single object instead of considering that all of its elements are individual objects.

5. Testing TM Performance

In this section, we present the performance results obtained with TMunit on a 4-Quad-Core AMD Opteron Processor 8354 running at 2.2Ghz (16 cores). We tested TinySTM, TL2, word-based RSTM, and SwissTM. We did not include WSTM in the performance graphs because of a problem encountered with porting the inline assembly code to the target architecture. This issue did not prevent us

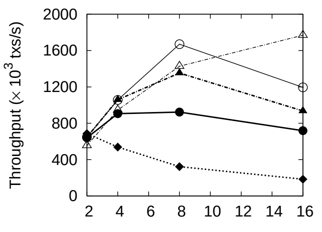
#	Test name	TL2	TinySTM			RSTM		SwissTM	WSTM	$\mathcal{E}$-STM
			ETL	CTL	WT	word	object			
	Safety tests									
1	opacity	✓	✓	✓	✓	✓	✓	✓	×	✓
2	linearizability	✓	✓	✓	✓	✓	✓	✓	✓	×
3	serializability	✓	✓	✓	✓	✓	✓	✓	✓	×
4	SGLA	×	×	×	×	×	×	×	×	×
	Unnecessary abort tests									
5	write-during-read-only	×	✓	✓	✓	×	✓	✓	✓	✓
6	invisible-write	✓	×	✓	×	✓	✓	✓	✓	×
7	false-sharing	✓	✓	✓	✓	×	✓	×	✓	×
8	virtual-world	×	×	×	×	×	×	×	×	×
	Isolation tests									
9	publication issue	×	×	×	×	×	×	×	×	×
10	dirty-read	✓	✓	✓	×	✓	✓	✓	✓	✓
11	zombie-transaction	✓	✓	✓	✓	✓	✓	✓	✓	✓
12	interference	✓	✓	✓	✓	✓	✓	✓	×	✓
13	granularity	✓	✓	✓	✓	✓	×	✓	✓	✓

Table 1: Results of our semantic test-suite obtained with the six TMs. Failures are denoted by the cross '×' while successes are denoted by the check-mark '✓'.

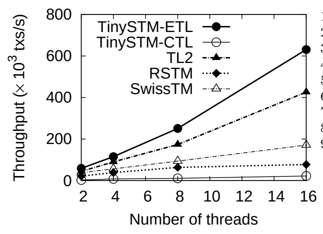
from executing the performance-insensitive semantic tests of Section 4 on another architecture. First, we investigate TM performance in extreme scenarios. Then, we compare the performance obtained for an existing benchmark implementation with the performance of TMunit running the corresponding specification of the benchmark.

5.1. Extreme scenarios

A powerful feature of TMunit is that it allows to quickly design workload to test specific scenarios or seldom exercised functionalities of the TMs. Here, we investigate the response of TMs in the face of extreme workloads that highlight the differences in TM designs. $\mathcal{E}$-STM is not tested here as there is no need for using elastic transactions in these tests and its normal transactions would have the same performance as TinySTM-ETL. These tests demonstrate that TM performance relies tightly on the workload used.



```
Definitions:
  SIZE=1024;
  arr[1..SIZE+100]=0;
  AD=<1..SIZE>;
Transactions:
  T := W(arr[AD]),
    {# k=[1..100]; R(arr[AD+k])};
Threads:
  P1, ..., P16 := T*
```



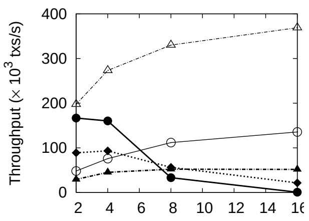
```
Definitions:
  N=1000;
  arr[1..16*N]=0;
Transactions:
  T1:={#k=[1..N]; W(arr[k])};
  T2:={#k=[N+1..N*2];W(arr[k])};
Threads:
  P1:=T1*;
  P2:=T2*; ...
```

Figure 10: **(Top)** Performance tests with write-once-read-many transactions and **(bottom)** with disjoint-writes transactions.

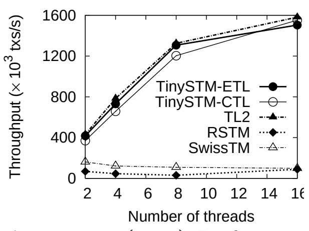
In Figure 10 (top), all threads execute a transaction composed of one write followed by a series of reads. This performance test is made such that the reads executed by one thread may access the same address as the one already written by another thread. This read-after-write pattern is expected to emphasize the circumstances in which commit-time locking is better suited than encounter-time locking. As expected, TinySTM-CTL and TL2 presents better throughput than TinySTM-ETL. SwissTM takes advantage of the extra version that can be accessed while a memory location is locked. An interesting observation is that word-based RSTM throughput is significantly lower than other STMs, despite using a commit-time locking strategy. The authors of RSTM have hinted as a possible reason the non-scalable libstdc++ exception mechanism used to trigger a roll-back upon abort.

Figure 10 (bottom) shows a scenario that highlights the cost of writes without contention. Threads perform series of writes to disjoint memory regions, which implies that there are no conflicts. One can observe that TinySTM-ETL scales best. TL2 suffers from using commit-time locking: it needs to check for every write whether the memory location has already been written by the same transaction, which requires a traversal of the write set. To limit the cost of this check, TL2 uses bloom filters but the overhead is still not negligible. SwissTM performs a double locking of the entries in the write set and, hence, is penalized when transactions write many memory locations. RSTM again shows scalability problems due to libstdc++. Finally, TinySTM-CTL suffers from the same problem as TL2 but was executed without the bloom filter optimization.

In Figure 11 (top), all threads execute a transaction composed of multiple write operations followed by multiple reads. In this performance test, operations executed by each transaction access consecutive addresses and generate much contention—a scenario where TM is typically less efficient than locking and contention management has great importance. Interestingly, SwissTM scales significantly



```
Definitions:
   SIZE=1024;
   arr[1..SIZE+100]=0;
   AD1=<1..SIZE>;
   AD2=<1..SIZE>;
Transactions:
   T :=
    {# k=[1..100]; W(arr[AD1+k])},
    {# k=[1..100]; R(arr[AD2+k])};
Threads:
   P1, ..., P16 := T*
```



```
Definitions:
   SIZE=256;
   arr[1..SIZE]=0;
Transactions:
   T1 := {#k=[0..15] {#j=[1..15];
    R(arr[k*16+j])}}; W(arr[0]);
   T2 := {#k=[0..15] {#j=[1..15];
    R(arr[k*16+j])}}; W(arr[16]);
Threads:
   P1 := T1*;
   P2 := T2*; ...
```

Figure 11: **(Top)** Performance tests with write-many-read-many transactions and **(bottom)** with false-sharing transactions.

better than the other TMs. To better understand the reason for this behavior, we have also experimented with TinySTM-ETL when (1) allowing transactions to read the previous version of locked memory locations by peeking into the write set of the lock owner, as in multi-version LSA (TinySTM-1v), and (2) activating TinySTM's built-in "priority" contention manager (TinySTM-CM). Each of these mechanisms provide noticeable improvement on this extreme workload. The remaining optimizations consist in choosing the right tuning parameters, as was studied in [14]. In particular, having each lock protect a set of consecutive memory locations (typically the size of a cache line) improves the performance because it causes fewer cache invalidations and reduces the number of compare-and-swap operations, as the lock needs to be acquired only once for several consecutive memory addresses.

To observe the influence of having a lock to protect multiple consecutive addresses (i.e., false-sharing as mentioned in Section 4.3), we have created a workload in which threads can only conflict if there is false sharing. In Figure 11 (bottom) each thread reads a series of consecutive addresses and writes a single, distinct address. To avoid undesirable memory effects, each write accesses an address on a distinct cache line next to addresses read by the other threads. This example may trigger false sharing: upon writing, threads acquire more addresses than necessary, including addresses that have been read by concurrent transactions, and produce unnecessary aborts. Indeed, we observe that both SwissTM and word-based RSTM are subject to false sharing, while other implementations are not.

5.2. TMunit *vs. hand-crafted benchmark*

Here, we compare an existing micro-benchmark, a linked list implementation of an integer set, to its corresponding TMunit specification (see Listing 3). The benchmark initially inserts a given number of elements in the linked list. Then, each thread starts executing and performs a series of *search* and *update* transactions (alternating inserts and removals to maintain the size of the list roughly unchanged during the whole execution) according to a given probability. A common problem in performance tests is the overhead introduced by the evaluation framework. As mentioned in Section 2 and to avoid this overhead, TMunit can translate the specification into C code to be compiled before execution. Here, we motivate this choice by comparing the results obtained using the generated technique against the results of the existing benchmark written in C code "by hand", denoted by native.

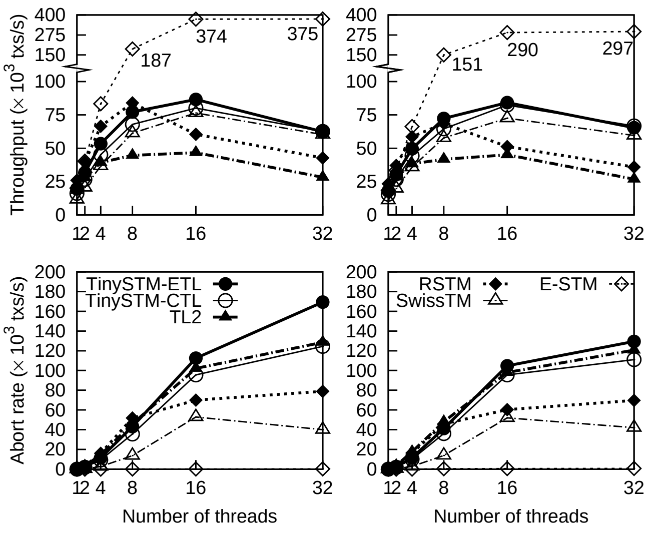
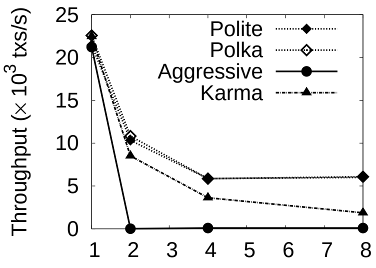


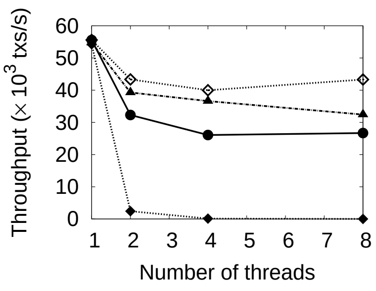
Figure 12: Performance comparison of the native intset benchmark **(left)** and the intset TMunit specification **(right)**. The commit rate is shown above and the abort rate below. Note that in order to distinguish curves other than $\mathcal{E}$-STM curve, the graphs for commit rates have different scales above and below 100 thousand tx/s.

Figure 12 presents the throughput (top) and the abort rate (bottom) of the linked list with an update transaction probability of 20%. A first observation is that the generated version presents results (commit and abort rates) very similar to the results of the hand-crafted benchmark. We have also executed the workload on the interpreted automaton and observed a 65% decrease in throughput with respect to the throughput of hand-crafted benchmark. This clearly motivates the need for the generated automaton.

$\mathcal{E}$-STM has also been tested here because the integer set operations can benefit from the elastic transactions. Actually, Figure 12 confirms that $\mathcal{E}$-STM performs best. $\mathcal{E}$-STM performance is followed by TinySTM-ETL, TinySTM-CTL and SwissTM, respectively. Word-based RSTM executes slower than other STMs again due to the scalability problems of libstdc++ that was pointed out in Section 5.1. The reason why TL2's performance is lower than TinySTM can be explained by the differences in timestamp management: TL2 does not perform dynamic snapshot extension as explained in Section 4.3. We can clearly see that $\mathcal{E}$-STM is not subject to contention as op-



```
Definitions:
  SIZE = 256;
  m[0..SIZE−1] = 0;
Transactions:
  T :=
     {# k=[0..SIZE−1]:
       R(m[k]), W(m[k]) };
Threads:
Th := T*;
```



```
Definitions:
 SIZE=1024;
 arr[1..SIZE+100]= 0;
 AD1=<1..SIZE>;
 AD2=<1..SIZE>;
Transactions:
 T :=
    {# k=[1..100]:
         W(arr[AD1+k])},
    {# k=[1..100]:
         R(arr[AD2+k])};
Threads:
 P1, ..., P16 := T*;
```

Figure 13: Performance variation of contention managers depending on the workload. **(Top)** long read-write workload. **(Bottom)** write-many-read-many workload.

posed to other STMs (even for more threads than processors there is no performance drop) as it uses elastic transactions that ensure the minimum guarantees to satisfy the correctness of integer set operation.

6. Testing CM Policy

In this section, we present the performance impact of contention management policy. A Contention Manager (CM) is a module that indicates how to solve conflicts depending on which transactions should take resolution actions and which should continue. Recently, the TM community has proposed multiple CMs [22, 51, 53]: some abort one of the conflicting transactions depending on which transaction detects the conflict first (Aggressive, Passive), some use exponential backoff time before resuming (Polite, Polka), and some rely on priorities based on the amount of past accesses (Karma, Polka, Eruption), priorities based on the amount of past aborts (Retry), or priorities based on time spent (Timestamp, Greedy).

6.1. Performance variations

Here we test several CMs we could find in the literature. To this end, we generated a code compatible with the object-based version of RSTM [10] where each access to a memory word was treated as an access to an object. RSTM is implemented so that it can be easily parameterized to run with one of these CMs. The results we obtained on an 8-core Intel Xeon CPU X5365 running at 3.00GHz are depicted in Figure 13.

In the long read-write workload of Figure 13 (top) every memory location accessed is a source of conflict and all transactions access the locations in the same order. So a backing off policy, as in Polite and Polka, provides a first-come-first-commit kind of ordering between transactions and allow progress. Clearly, the Aggressive policy results in continuous aborts and cannot provide any progress.

In the write-many-read-many workload depicted in Figure 13 (bottom), transactions forcing other transactions to backoff may conflict later on and backoff in turn, hence blocking for a while. Since the workload is designed to be highly contended, such situations are most likely, thus a simple backoff policy like Polite slows performance down. In contrast, Polka, which incorporates an effort-based priority scheme as in Karma, copes with this problem and performs well. Finally, Aggressive also avoids blocking.

6.2. Livelocks

As claimed in [53], the Passive CM may suffer livelocks. We experiment the existence of this issue by providing a dedicated workload that can be reproduced easily on other TM/CM using TMunit.

The *bank-benchmark* test, specified in Listing 4, uses a classical scenario where one thread computes the sum of the balances of 1024 to 8192 accounts of a bank (long read-only transaction) while the other threads concurrently perform transfers (short update transactions). In TM designs with invisible reads and no fair contention management, updates conflicting with long read-only transactions may lead to failed validation. The problem is illustrated in Figure 14 (left), where short *transfer* transactions prevent the long *balance* transactions from committing. Figure 14 (center) shows the result when using Passive CM in TinySTM while Figure 14 (right) presents the result with the built-in priority-based contention manager. With Passive CM, throughput drops to 0 when the number of threads performing transfers reaches 3. We inspected the corresponding TMunit trace to make sure that the cause was the problem described in Figure 14 (left). As expected, when using the Retry CM, the throughput is almost independent of the number of threads performing transfers. Unlike Passive CM, Retry CM ensures progress.

To conclude, we have experimentally demonstrated the initial thoughts under which some CM are more progress-friendly than others. Thanks to TMunit these scenarios are easily reproducible for further testings and may outline similar liveness issues in future implementations.

```
Definitions:                                                  // variables and constants
  NB = 8192;                                              // number of accounts (constant)
  a[1 .. NB] = 0;                                          // memory range for accounts
  SRC = <1 .. NB>;                               // random value (constant) in range 1. . .NB
  DST = <1 .. NB>;                               // random value (constant) in range 1. . .NB

Transactions:                                            // specification of transactions
  T1 := R(a[SRC]), R(a[DST]), W(a[SRC]), W(a[DST]) ;                       // transfer
  T2 := {# k = [1 .. NB] : R(a[k]) } ;                                 // compute balance

Threads:                                                    // specification of threads
  P1 := T2*;
  P2, P3, P4, P5, P6, P7, P8 := T1*;
```

Listing 4: Specification of the bank benchmark that exhibits lack of progress (for NB=8192 accounts and 8 threads).

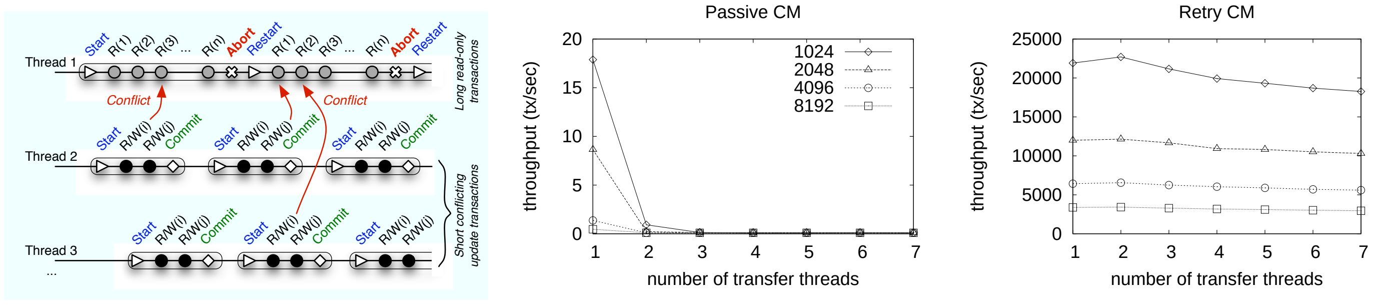


Figure 14: Bank account benchmark suffering from the lack of progress when Passive CM is used. Scenario is given on the left, while the throughput from Passive and Retry CMs (for 1024 to 8192 bank accounts) appear on the center and right-hand side.

7. Related Work

Here, we overview work related to the testing of transactional memory. Those frameworks test concurrent applications in different ways ranging from exhaustive tests to verify a given program, randomized tests necessary to evaluate average performance and deterministic tests that can be replayed to identify the causes of potential issues.

7.1. Testing the semantics of concurrent programs

The tightest related body of work relies on the dynamic testing technique that collects information about concurrency defects by executing the software itself as opposed to the static technique that analyzes source code to extract information about possible defects such as data races, atomicity violations or deadlocks.

7.1.1. Randomized tests

Some non-deterministic testing approaches target the detection of data races, e.g., [50], some other target the detection of atomicity violations, e.g., [37, 38]. Since the tested schedules cannot be exhaustive, existing frameworks [5, 54] try to increase the schedule coverage by inserting sleep of yield function to produce scheduling variations. Finally, in [40], the authors use pseudo-random tests to verify whether the execution matches the consistency specified by the TM.

Unfortunately, non-deterministic tests are inherently irreproducible thus they better apply to test a program as a whole rather than to identify precise problematic scenarios. In contrast here, our goal is to identify the pathological schedules that make TM to fail or to violate its semantics. Hence, the reproducibility of the testing is crucial for our needs.

7.1.2. Deterministic tests

A pioneering work on deterministic testing of concurrent programs is by Carver et al. [55], which is based on generating deterministic schedules and replaying them for testing. In their approach, the interleaving points of the schedules are the synchronization operations (e.g., mutex or semaphore accesses) and thus they can generate a schedule using a sequence of synchronization operations of the tested program. The advances in capturing possible thread schedules of a concurrent program result in various concurrent application schedules that can be replayed deterministically for concurrency defects. ConTest [13] aims at reducing the schedule search space using some coverage metric whereas CHESS [42] models all synchronization operations as interleaving points, and restricts the schedule search space (i) by using fair schedules and (ii) by limiting the number of preemptions.

Verisoft [16] proposes to use systematic state-space exploration with a model-checker applied directly to the program. In this approach the state-space corresponds to the combined behavior of all concurrent components of the program. This approach eliminates the redundant state transitions based on independence of transitions.

We should mention MultithreadedTC [47] and ConAn [35] as unit testing frameworks that use an external clock, similar to the scheduler thread of TMUnit, to synchronize threads and enforce deterministic schedules for Java programs.

Unlike aforementioned proposals that generate and test the schedule search space of concurrent programs, we only focus on schedule search space of TM programs. More precisely, our goal is not to detect races on a TM metadata but rather detecting data races between data items accessed by TMs. Hence, our approach inherently limits the number of schedules to explore which makes deterministic testing feasible. It is noteworthy that verification of TM correctness [19–21, 44] are based on a formal specification of TM algorithms and not on the actual TM code which makes the testing process more difficult. Other testing tools [27] help on the step-by-step debugging of TMs but are out of the scope of this paper.

7.2. Testing the performance of concurrent programs

Shared-memory management in concurrent programs have been extensively evaluated as demonstrated by the numerous multithreaded benchmarks such as SPLASH-2 [56], PARSEC [6], SPEComp [4]. Among these benchmarks SPEComp is specialized on high performance computing applications, while SPLASH-2 and PARSEC include applications from different domains. In addition, there exist other multithreaded benchmark suites specialized to a given application domain like BioParallel [31] dedicated to bioinformatics, ALPBench [39] dedicated to multimedia and MineBench [43] dedicated to data mining. The sole work we know of that attempted to port such applications to transactional memory, is by Chung et al. [9], however, it uses lock elision which is reported to be unsafe [7, 9].

In contrast with the aforementioned benchmarks, TM benchmarks are very appealing as they can test any TM as long as it fulfills the given interface requirements. TM micro-benchmarks that are based on common data-structure have been used to evaluate TMs at the early stages. TL2 has been implemented on red-black trees [11] while TinySTM has been implemented on linked-lists [14]. DSTM has been implemented on both linked-list and red-black trees [28]. Microbench is a micro-benchmark suite that compares recent lock-free and lock-based implementation of common data structures (including skip list, hash table) to some pluggable TM [15]. Those benchmarks tests few different type of transactions that modifies or searches the data-structure.

As further attempts to mimic realistic settings, more complex TM benchmarks have been proposed. STM-Bench7 [24] extends the OO7 benchmark that was used to evaluate databases. This benchmark executes several types of workloads that access a large graph of updatable elements. Generally, the transactions it generates are more complex than in micro-benchmarks as the transactions can be very long without necessarily accessing common elements of the data structure. As a side-effect, it consumes more memory than micro-benchmarks. Haskell STM benchmark suite [46] provides both small realistic applications and several micro-benchmarks written in Haskell. Wormbench [57] intends to provide a synthetic workload generating applications that can stress specific designs and implementation aspects of TM systems including TM library, code instrumentation and compiler optimizations.

Finally TM macro-benchmarks are higher level benchmarks that often integrates real-world applications. The most widely used macro-benchmark suite STAMP [8] comprises eight different parallel applications as, for example, an online multi-threaded reservation service or a Delaunay triangulation algorithm. Lee-TM [3] is a benchmark suite based on the Lee's routing algorithm and provides implementations of the algorithm in different granularities for both lock-based and transactional versions. RMS-TM [32] provides 4 benchmarks with nested transactions, memory management operations and I/O calls inside transactions from recognition, mining and synthesis domains.

Although those macro-benchmarks are invaluable for TM developers, they are limited by the number and type of applications available: extending the benchmark space requires to fully implement new applications.

8. Conclusion

This paper presents the first to date extensible testbed for transactional memory. This work relies on the novel TMunit framework that provides a domain specific language to simplify the writing of Transactional Memory (TM) tests. TMunit is efficient, thanks to its automated code generation tool, and provides a workload specification language that is simple and powerful. TMunit is already available online and comes with its test-suite, hence we hope that other researchers will contribute in extending this test-suite. For instance, TMunit is currently being used by AMD in the development of their advanced synchronization facility [1].

We have performed extensive experiments on six TMs and various CMs to compare their behaviors with theoretical expectations. We identified TMs violating SGLA, opacity and virtual-world consistency and CMs violating progressiveness.

Apart from the fact that TM developers can use TMunit for verifying the behavior and performance of their TM, we envisage other uses. For example, the application developer can verify if her/his assumptions are satisfied by the underlying TM. This could be used to select the most efficient TM variant that still satisfies the application requirements.

Acknowledgments

We are grateful to Michael Spear who pointed out some causes for the obtained RSTM results, Patrick Marlier for his help on adapting WSTM for TMunit, and Aleksandar Dragojević for providing us with early SwissTM releases. This research is conducted within the FP7 Integrated Approach to Transactional Memory on Multi- and Many-core Computers (VELOX) project (ICT-216852) supported by the European Commission and within the Swiss National Foundation under grant 200021-118043.

References

[1] Evaluation of the advanced synchronization facility (ASF), `http://forums.amd.com/devblog/blogpost.cfm?threadid=118419&catid=317` (2009).

[2] M. Abadi, A. Birrell, T. Harris, M. Isard, Semantics of transactional memory and automatic mutual exclusion, SIGPLAN Not. 43 (1) (2008) 63–74.

[3] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, I. Watson, Lee-TM: A non-trivial benchmark for transactional memory, in: ICA3PP, 2008, pp. 196–207.

[4] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, B. Parady, SPEComp: a new benchmark suite for measuring parallel computer performance, in: WOMPAT, 2001, pp. 1–10.

[5] Y. Ben-Asher, E. Farchi, Y. Eytani, Heuristics for finding concurrent bugs, in: IPDPS, 2003, p. 288.1.

[6] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: PACT, 2008, pp. 72–81.

[7] C. Blundell, E. Lewis, M. Martin, Deconstructing transactional semantics: The subtleties of atomicity, in: In The 4th Workshop on Duplicating, Deconstructing, and Debunking, 2005.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IISWC, 2008, pp. 35–46.

[9] J. Chung, H. Chafi, C. Minh, A. McDonald, B. Carlstrom, K. Kozyrakis, C. Olukotun, The common case transactional behavior of multithreaded programs, in: HPCA, 2006, pp. 266–277.

[10] L. Dalessandro, V. J. Marathe, M. F. Spear, M. L. Scott, Capabilities and limitations of library-based software transactional memory in C++, in: TRANSACT, 2007.

[11] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: DISC, 2006, pp. 194–208.

[12] A. Dragojevic, R. Guerraoui, M. Kapalka, Stretching transactional memory, in: PLDI, 2009, pp. 155–165.

[13] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, S. Ur, Framework for testing multi-threaded java programs, Concurrency and Computation: Practice and Experience 15 (3-5) (2003) 485–499.

[14] P. Felber, C. Fetzer, T. Riegel, Dynamic performance tuning of word-based software transactional memory, in: PPoPP, 2008, pp. 237–246.

[15] P. Felber, V. Gramoli, R. Guerraoui, Elastic transactions, in: DISC, vol. 5805 of LNCS, 2009, pp. 93–107.

[16] P. Godefroid, Model checking for programming languages using verisoft, in: POPL, 1997, pp. 174–186.

[17] V. Gramoli, D. Harmanci, P. Felber, Toward a theory of input acceptance for transactional memories, in: OPODIS, vol. 5401 of LNCS, 2008, pp. 527–533.

[18] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers Inc., 1992.

[19] R. Guerraoui, T. A. Henzinger, B. Jobstmann, V. Singh, Model checking transactional memories, in: PLDI, 2008, pp. 372–382.

[20] R. Guerraoui, T. A. Henzinger, V. Singh, Nondeterminism and completeness in transactional memories, in: CONCUR, 2008, pp. 21–35.

[21] R. Guerraoui, T. A. Henzinger, V. Singh, Software transactional memory on relaxed memory models, in: CAV, 2009, pp. 321–336.

[22] R. Guerraoui, M. Herlihy, B. Pochon, Polymorphic contention management, in: DISC, 2005, pp. 303–323.

[23] R. Guerraoui, M. Kapałka, On the correctness of transactional memory, in: PPoPP, 2008, pp. 175–184.

[24] R. Guerraoui, M. Kapałka, J. Vitek, STMBench7: A benchmark for software transactional memory, SIGOPS Oper. Syst. Rev. 41 (3) (2007) 315–324.

[25] D. Harmanci, P. Felber, V. Gramoli, C. Fetzer, TMunit: Testing software transactional memories, in: TRANSACT, 2009.

[26] T. Harris, K. Fraser, Language support for lightweight transactions, in: OOPSLA, 2003, pp. 388–402.

[27] M. Herlihy, Y. Lev, tm_db: A generic debugging library for transactional programs, in: PACT, 2009, pp. 136–145.

[28] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, Software transactional memory for dynamic-sized data structures, in: PODC, 2003, pp. 92–101.

[29] M. Herlihy, J. M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. on Programming Languages and Systems 12 (3) (1990) 463–492.

[30] D. Imbs, J. R. González de Mendvil, M. Raynal, Virtual world consistency: A new condition for STM systems, in: PODC, 2009, pp. 280–281.

[31] A. Jaleel, M. Mattina, B. Jacob, Last level cache (LLC) performance of data mining workloads on a CMP – a case study of parallel bioinformatics workloads, in: HPCA, 2006, pp. 88–98.

[32] G. Kestor, S. Stipic, O. S. Unsal, A. Cristal, M. Valero, RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications, in: TRANSACT, 2009.

[33] J. Larus, R. Rajwar, Transactional Memory, Morgan & Claypool Publishers, 2006.

[34] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, M. Olszewski, Anatomy of a scalable software transactional memory, in: TRANSACT, 2009.

[35] B. Long, D. Hoffman, P. Strooper, Tool support for testing concurrent Java components, IEEE Trans. Softw. Eng. 29 (6) (2003) 555–566.

[36] J. Lourenço, G. Cunha, Testing patterns for software transactional memory engines, in: ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, 2007, pp. 36–42.

[37] S. Lu, J. Tucek, F. Qin, Y. Zhou, AVIO: Detecting atomicity violations via access interleaving invariants, in: ASPLOS, 2006, pp. 37–48.

[38] B. Lucia, J. Devietti, K. Strauss, L. Ceze, Atom-aid: Detecting and surviving atomicity violations, in: Int'l Symposium on Computer Architecture, 2008, pp. 277–288.

[39] R. Man-Lap Li Sasanka, S. Adve, Y.-K. Chen, E. Debes, The ALPBench benchmark suite for complex multimedia applications, in: IISWC, 2005, pp. 34–45.

[40] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, K. Olukotun, Testing implementations of transactional memory, in: PACT, 2006, pp. 134–143.

[41] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, A. Welc, Practical weak-atomicity semantics for Java STM, in: SPAA, 2008, pp. 314–325.

[42] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, I. Neamtiu, Finding and reproducing heisenbugs in concurrent programs, in: OSDI, 2008, pp. 267–280.

[43] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, A. Choudhary, Minebench: A benchmark suite for data mining workloads, in: Int'l Symp. on Workload Characterization, 2006, pp. 182–188.

[44] J. O'Leary, B. Saha, M. R. Tuttle, Model checking transactional memory with Spin, in: ICDCS, 2009, pp. 335–342.

[45] C. H. Papadimitriou, The serializability of concurrent database updates, J. ACM 26 (4) (1979) 631–653.

[46] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, M. Valero, The limits of software transactional memory (STM): Dissecting haskell STM applications on a many-core environment, in: Conference on Computing Frontiers, 2008, pp. 67–78.

[47] W. Pugh, N. Ayewah, Unit testing concurrent software, in: ASE, 2007, pp. 513–516.

[48] T. Riegel, P. Felber, C. Fetzer, A lazy snapshot algorithm with eager validation, in: DISC, 2006, pp. 284–298.

[49] T. Riegel, C. Fetzer, H. Sturzrehm, P. Felber, From causal to z-linearizable transactional memory, Tech. Rep. RR-I-07-02.1, Université de Neuchâtel, Switzerland (2007).

[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A dynamic data race detector for multithreaded programs, ACM Trans. Comput. Syst. 15 (4) (1997) 391–411.

[51] W. N. Scherer III, M. L. Scott, Contention management in dynamic software transactional memory, in: Workshop on Concurrency and Synchronization in Java Programs, 2004.

[52] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, B. Saha, Enforcing isolation and ordering in STM, SIGPLAN Not. 42 (6) (2007) 78–88.

[53] M. Spear, L. Dalessandro, V. Marathe, M. Scott, A comprehensive strategy for contention management in software transactional memory, in: PPoPP, 2009, pp. 141–150.

[54] S. D. Stoller, Testing concurrent Java programs using randomized scheduling, in: Workshop on Runtime Verification, vol. 70(4), 2002, pp. 142–157.

[55] K.-C. Tai, R. H. Carver, E. E. Obaid, Debugging concurrent Ada programs by deterministic execution, IEEE Trans. Softw. Eng. 17 (1) (1991) 45–63.

[56] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The splash-2 programs: characterization and methodological considerations, in: Int'l Symposium on Computer architecture, 1995, pp. 24–36.

[57] F. Zyulkyarov, S. Cvijic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, WormBench - A configurable workload for evaluating transactional memory systems, in: MEDEA Wokshop, 2008, pp. 61–68.