TMUNIT: Testing Software Transactional Memories

Derin Harmanci Pascal Felber
University of Neuchâtel
Switzerland
{derin.harmanci,pascal.felber}@unine.ch

Vincent Gramoli
EPFL & University of Neuchâtel
Switzerland
vincent.gramoli@epfl.ch

Christof Fetzer
Dresden University of Technology
Germany
christof.fetzer@tu-dresden.de

Abstract

Transactional memory (TM) is expected to become a widely used parallel programming paradigm for multi-core architectures. To reach this goal, we need tools that do not only help develop TMs, but also test them and evaluate them on a wide range of workloads. In this paper, we introduce a novel tool, TMUNIT, to assist researchers in designing and optimizing TMs. TMUNIT provides a domain-specific language for specifying workloads, and tests the performance and semantics of TMs. TMUNIT is freely available online. It comes with a test suite that compares the performance of TMs and explain their differences using semantics tests that outlines behavioral characteristics.

1. Introduction

The transactional memory (TM) paradigm is appealing for programming parallel applications thanks to its simplicity. This paradigm supersedes the existing lock-based paradigm in two ways. First, it allows more concurrency than coarse-grained locking strategies. Second, it composes well as opposed to fine-grained locking, and is much less error-prone. Our aim is here to assist TM developers and users in testing both performance and semantics of TM implementations.

For the past few years, researchers from industry and academia have devoted many efforts on studying transactional memory semantics and highlighting important open questions, such as: how should a transaction behave in presence of concurrent non-transactional accesses (in a weak-atomicity model) [1], should a transaction abort even though its commit would not violate consistency [7], or 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 [15]. These questions outline the essential role of specific interleavings of conflicting operations in transactions. Various examples have been described as code fragments that expose anomalies when transactional operations are executed in a certain order [1, 7, 9, 15, 18, 20].

There is a crucial need to evaluate the semantics of transactional memory in these specific circumstances. Evaluating a transactional memory can help TM developers analyze the behavior of their implementation and identify bugs, but it may also help TM users test their application workload on specific TM implementations with an associated memory model. The only existing testing tools for transactional memory target step-by-step debugging [13] or safety verification [14]. To the best of our knowledge, no tool can perform comprehensive semantic tests of a TM.

Testing parallel programs in a reproducible manner is not an easy problem. 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. The impact is even more significant in transactional memory where threads communicate faster than in message passing model. Typically, it would be desirable to test the performance and semantics of a TM using *unit tests*.

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 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 unit 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 propose TMUNIT, a novel tool that can help transactional memory researchers evaluate the semantics, validate the implementation, and improve the performance of TMs.

Related work. We distinguish two major groups of experimental evaluations for TMs: performance evaluation and semantic evaluation (debugging/testing/verification). The biggest challenge for performance evaluation of TMs is to propose benchmarks that are *(i)* precise enough to emphasize TM characteristics, but also *(ii)* realistic enough to match the behavior of common applications.

To achieve these goals, many authors [3, 6, 12] evaluate performance of TMs using micro-benchmarks that access simple data structures, such as linked lists or red-black trees. However, more elaborate and configurable evaluation frameworks were proposed. STMBench7 [10] presents such a framework composed of a complex data structure and update/traversal primitives on this data structure. Wormbench [21] aims at representing applications by a series of complex atomic operations, but it does not allow operation interleavings to be controlled by the user.

Some other studies put more stress on realistic workloads. The major framework in this direction is STAMP [16]. There is also the Haskell benchmark suite [17] which is a mixture of micro-benchmarks and some real world applications. While those studies are invaluable to assess TM performance, they are limited by the number and type of benchmarks.

For the semantic evaluation of TMs we see mainly two types of studies. The first type of studies (e.g., [14]) perform non-deterministic execution of randomized workloads and verify their correctness post-execution. The second type of studies (e.g., [13]) mainly studies the debugging of TMs.

Contributions. We have designed a tool, TMUNIT, that provides a domain-specific language dedicated to TM workloads. This language aims at being simple and expressive. It allows users to rapidly specify benchmarks to evaluate and compare TMs, as well as unit tests to validate the behavior of a specific implementation. Like other TM evaluation frameworks, 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.

TMUNIT has been successfully tested on five word-based TM implementations:[1] WSTM [11], TinySTM [6], TL2 [3], RSTM [2], and SwissTM [4]. The results we obtained from TMUNIT indicate that TM performance relies tightly on the chosen workload and may vary drastically. TMUNIT semantic tests revealed interesting insights into the implementation of the TMs, notably that WSTM violates opacity [9] unless explicit validation is performed by the programmer.

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. The current release of TMUNIT can be found at `http://www.tmware.org/tmunit`.

Roadmap. In Section 2, we present the architecture of TMUNIT. In Section 3, we describe our domain-specific language and give some examples. Sections 4 and 5 test, respectively, the semantics and the performance of the TMs. Finally, Section 6 concludes the paper.

2. TMUNIT Architecture

In this section, we briefly describe the main components of TMUNIT and explain how they interact (see Figure 2).

As mentioned earlier, TMUNIT executes a workload on a dedicated TM. This workload corresponds to a synthetic description written in a domain-specific language (see Section 3). TMUNIT can either execute this workload dynamically or convert it into a compiled application to reduce the runtime overheads. TMUNIT also records performance statistics and test results.

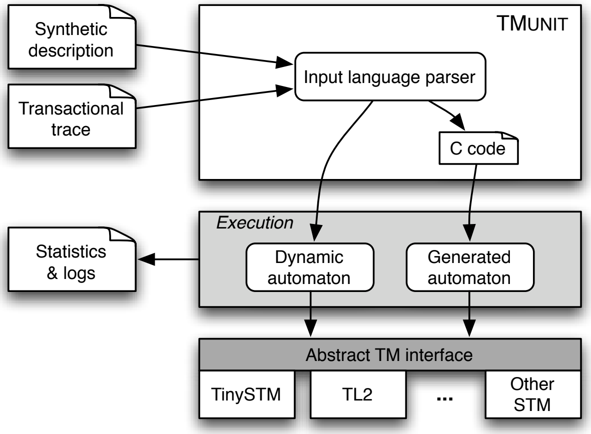


Figure 2: Architectural overview of TMUNIT.

More precisely, a test is modeled as an automaton performing transactional operation calls to an TM implementation. TMUNIT uses an abstract TM interface to map transactions onto the underlying TM. TMUNIT can use two different automata: an *interpreted* automaton or a *generated* automaton. The interpreted automaton is produced when the configuration file representing the test specification is dynamically interpreted for direct execution. The generated au-

[1] While we use the term "TM" for generality, we only consider here *software* transactional memories (STMs).

tomaton is produced when the configuration file is translated to a different programming language.

Currently, the generated automaton only produces C code for word-based TM implementations. While the interpreted automaton can be used to interpret and execute the configuration file in a one-step process, the generated automaton executes more efficiently, 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. The decision of the execution mode is taken by the user. Either automaton may output a trace of the execution and detailed performance statistics.

3. A Language for TM Workloads

Evaluating TMs requires to test their behavior not only in response of minimal workloads (by unit testing) but also in response of more complex workloads (by performance testing). In this section, we give a high level description of TMUnit's generic language to specify TM workloads.

The language has been designed to be simple enough to specify transactions and schedules in an abstract way as usually found in academic papers (i.e., using sequence of reads and writes on variables), and expressive enough to reproduce classical transactional benchmarks like operations on linked lists.

```
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:

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.

3.1 Unit test specification

A *unit test* refers to a simple reproducible test that exercises a specific aspect of the TM implementation, e.g., whether a read-after-write returns the last value written or whether dirty reads do not occur. Unit tests are generally used for the sake of integration following a bottom-up approach [5]. Here, we consider a specific kind of unit 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. Listing 2 illustrates our domain-specific language on the *zombie transactions* example of [1].[2] 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 [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. Undeclared variables are assumed to be integers. One can also use thread-local variables for specifying complex patterns in transactions such as loops or conditionals. 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). The initial value of a variable can be set in the definition section. Variables that are not explicitly initialized are set to 0. Variable names with only capital letters are considered as constants and their value cannot be modified.

[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.

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>)`. Write operations are performed by the underlying TM. 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. Evaluation follows the usual operator priorities.

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.

Schedules and assertions. Schedules specify a pre-defined interleaving of the transactions for testing or debugging a TM. 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 each transaction executes in its own thread, but only one thread is active at each step of the schedule. Multiple schedules can be specified but only one will be executed at a time; this schedule can be specified using command-line parameters.

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.

In schedule execution mode TMUNIT creates a thread per each defined transaction and an additional *scheduler* thread. It is the scheduler thread which 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 the 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 the first scheduled transaction. The thread owning the token executes until the next barrier it encounters (if there is none it executes until the end of the corresponding transaction) and then passes the token back to the scheduler thread. Upon receipt the scheduler thread passes the token to the next scheduled transaction's thread. This continues until the end of the defined schedule.

An example schedule for the "zombie transactions" scenario [1] 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.

3.2 Performance test specification

Performance tests are generally longer than unit tests since they execute complex schedules to measure the performance of a TM. More precisely, they use randomization and loops to test a large set of schedules. 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 (see Listing 3).

Randomness and loops. To implement realistic performance tests, our language provides powerful constructs such as random executions and loops. Randomness allows users to run a set of patterns non-deterministically while loops can repeat the same pattern multiple times. 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 and a pointer to the next node). This transaction mimics the search for a random element in

```
 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]) },      // T1: search element
15          R(m[2*NB]) ;
16   T2 := {? [_f == 0] :                               // T2: 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.

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 upon node insertion is that the new node is not shared until commit time; in contrast there are three writes upon removal 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.

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.

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" [7] to "publication" [15] 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[3] and apply them on several TM implementations. The summary of the results is given in Table 1.

We experimented in this paper with five word-based TM implementations all using invisible reads, i.e., transactions are not aware that others might be reading a shared memory location. WSTM [11] is an early TM implementation that associates version numbers to memory locations and uses a 2-phase-locking strategy at commit-time. TinySTM [6] is a C implementation of LSA [19] (single-version variant), the first time-based STM algorithm. It relies on a global clock to guarantee that reads are always consistent while avoiding the quadratic cost of incremental validation. The versions we used are *(i)* encounter-time locking (ETL) that makes writes visible to other transactions immediately and *(ii)* commit-time locking (CTL) keeps writes invisible until commit time. For isolation tests, we also used the write-through (WT) version that updates data in memory immediately upon write, instead of commit time as other versions. Finally, to highlight liveness problems, we experimented with and without contention management. TL2 [3] also uses a time-based algorithm, but unlike LSA it does not dynamically "extend" validity snapshots at runtime. By default, TL2 uses a commit-time locking strategy. The word-based version of RSTM [2] follows a design similar to TL2. Finally, SwissTM [4] is a C++ implementation of LSA with dynamic snapshot extensions and an encounter-time locking strategy. The main difference with TinySTM is that it allows reading old versions of locked memory locations, as in multi-version LSA. It also integrates more sophisticated contention management mechanisms.

[3] All the semantic tests covered in this paper can be found in the latest release of TMUNIT on `http://tmware.org/tmunit`

#	Test name	TL2	TinySTM			RSTM	SwissTM	WSTM
			ETL	CTL	WT			
	Safety tests							
1	opacity	S	S	S	S	S	S	F
2	linearizability	S	S	S	S	S	S	S
3	serializability	S	S	S	S	S	S	S
4	SGLA	F	F	F	F	F	F	F
	Unnecessary abort tests							
5	write-during-read-only	F	S	S	S	F	S	S
6	invisible-write	S	F	S	F	S	S	S
7	false-sharing	S	S	S	S	F	F	S
	Isolation tests							
8	publication issue	F	F	F	F	F	F	F
9	dirty-read	S	S	S	F	S	S	S
10	zombie-transaction	S	S	S	S	S	S	S
11	interference	S	S	S	S	S	S	F

Table 1: Results of our semantic test-suite obtained with the five TMs. Failures are denoted by 'F' while successes are denoted by 'S'.

4.1 Safety tests

The safety tests are interesting to better understand the consistency criterion ensured by TMs. Some TMs ensure opacity [9], some other ensure serializability. As pointed out in [7], however, some serializable TMs may also be opaque.

Here, we chose four safety tests from the literature. The *opacity* test (1) comes from Figure 1 of [9] and was used to illustrate the difference between strong atomicity and opacity requirements.The *linearizability* test (2), taken from Figure 2 of [7], aims at showing that a serializable software TM may not be linearizable. The *serializability* test (3) from Figure 2 of [18] exhibits that a software TM is not serializable. Finally the *SGLA* test (4) given in Figure 11 of [15] indicates how single-global-lock-atomicity (SGLA) [15] can be violated.

TL2, TinySTM, RSTM, and SwissTM pass all the four critical tests we proposed. 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. Unexpectedly, however, WSTM successfully passes all the tests but the first one. This test allows us to conclude 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 below in the *interference* test, WSTM allows this to happen.[4]

4.2 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* [7] 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:
  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 3: **(Top)** Write-during-readonly test for which TL2 and RSTM abort while other TMs commit (TMUNIT gives the same trace for TL2 and RSTM, represented on the right-hand side). **(Bottom)** False-sharing test for which SwissTM and RSTM abort while the other TMs commit (the trace of SwissTM is given on the right-hand side).

We test the five TM implementations on three unnecessary abort tests. The *write-during-readonly* test (5) is presented in Figure 3 (top). RSTM and TL2 fail this test while TinySTM, SwissTM, and WSTM succeed. The reason for the success of TinySTM and SwissTM is because they both use the LSA validation extension mechanism while RSTM and TL2 do not.

The *invisible-write* test (6) comes from the Figure 1 of [7] and checks whether unnecessary aborts may occur when the write operation is made visible. TinySTM-ETL fails the *invisible-write* test because it locks the to-be-written address eagerly as soon as a write occurs.

Finally, the *false-sharing* test (7) described in Figure 3 (bottom) 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 and RSTM abort meaning they use a common lock on (at least) two consecutive addresses. Unlike other STMs, they are subject to false sharing.

4.3 Isolation tests

We tested four 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.[5]

Specifically, we performed the following isolation tests. The *publication* test (8) as described in Figure 1 of [15] outlines a possible difference between the TM semantics and the lock semantics of the Java memory model. The *dirty-read* test (9) and the *zombie transaction* (10) test have been

[4] A workaround is to explicitly validate the transaction after every read.

[5] We did not use the fences of RSTM that would counteract our schedules.

discussed earlier (see Figure 1 and Listing 2, respectively). Finally, the *interference* test (11), described in Figure 4, outlines a possible interference between two transactions.

As expected [15], all TMs fail the publication-test (8). 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 (9), other TMs succeed. The reason is that, with the write-through strategy, updates are directly written to memory upon encountering a write operation, and possibly 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 (10). First, all the TMs that use the write-back strategy defer modification to commit time so that transaction T_2 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.

```
Definitions:                          ...
  x=0; y=0; _v=0; _w=0;               [Th1:T1] R(y)
Transactions:                         [Th1:T1] R(y,1)
  T1 := R(x,_v), @L, R(y,_w);         [Th1:T1] Try C
  T2 := W(x,1), W(y,1);               [Th1:T1] A
Schedules:                            [Th1:T1] Terminates
  S := T1@L, T2, T1, [_v==_w];        Assertion [_v==_w] fails.
```

Figure 4: The interference test for which WSTM fails while other TMs succeed. WSTM trace appears on the right-hand side.

The role of *interference* test (11) is to check whether a writing transaction can interfere with a concurrently reading transaction. For TL2, TinySTM, RSTM, and SwissTM, 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 (this violation has been detected in Subsection 4.1), since opacity requires 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: $T1$ observing that `x` is different from `y` may provoke an infinite loop or an irrevocable external event, like missile firing [8].

4.4 Liveness test

We now demonstrate how TMUNIT can effectively be used for testing liveness. We present a problem similar to a livelock, where some transactions are repeatedly aborted by others and cannot progress.

The *bank-benchmark* test, specified in Listing 4, uses a classical scenario where one thread computes the sum of the balances of all 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 5 (left), where short *transfer* transactions prevent the long *balance* transactions from committing. Note that this scenario can happen with any obstruction-free TM and is not an indication of a "buggy" design. Most TMs incorporate some form of contention management, ranging from simple exponential backoff (TL2) to sophisticated strategies (TinySTM, SwissTM).

```
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 5 (center) shows the result when disabling contention management in TinySTM while Figure 5 (right) presents the result with the built-in priority-based contention manager. Without contention manager 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 5 (left). As expected, when using the contention manager, the throughput is almost independent of the number of threads performing transfers. Hence the contention manager ensures progress.

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, 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. Unfortunately we could not fix this problem in time, but it did not prevent us 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 re-

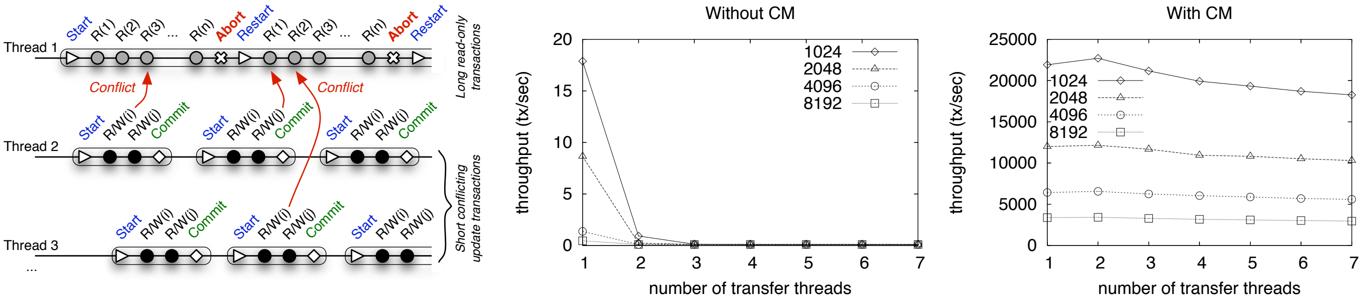
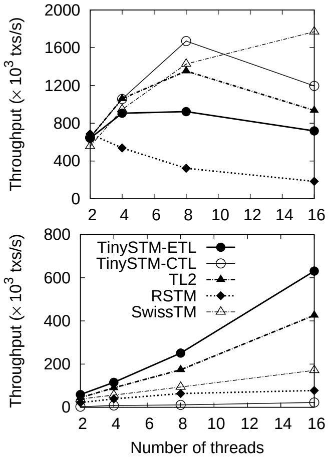


Figure 5: Illustration of the lack of progress of long read-only transactions in the bank benchmark (left). Comparison of the commit rate without (center) and with (right) contention manager for various numbers of bank accounts (ranging from 1024 to 8192) and threads performing transfers.

sponse of TMs in the face of extreme workloads that highlight the differences in TM designs. These tests also demonstrate that TM performance relies tightly on the workload used.



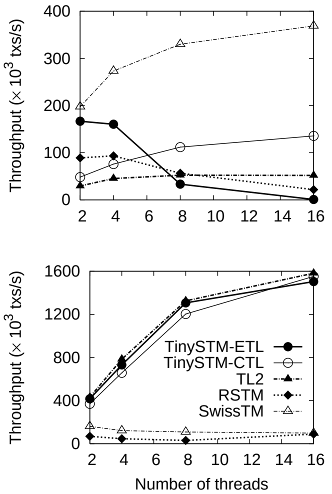
```
Definitions:
   SIZE=1024;
   arr[1..SIZE+100]
   AD=<1..SIZE>;
Transactions:
   T := W(arr[AD]),
     {# k=[1..100]; R(arr[AD+k])};
Threads:
   P1, ..., P16 := T∗
```

```
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 6: Performance tests with write-once-read-many transactions (top) and with disjoint-writes transactions (bottom).

In Figure 6 (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 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 6 (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++ library. Finally, TinySTM-CTL suffers from the same problem as TL2 but was executed without the bloom filter optimization.



```
Definitions:
   SIZE=1024;
   arr[1..SIZE+100];
   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];
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 7: Performance tests with write-many-read-many transactions (top) and disjoint-reads-write transactions (bottom).

In Figure 7 (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 better than the other TMs. To better understand the reason for this

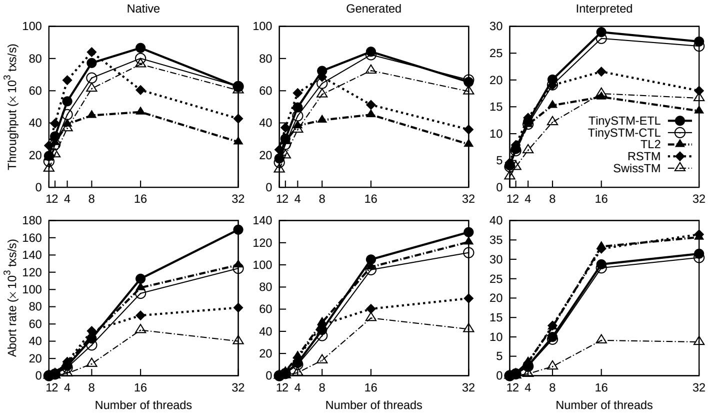


Figure 8: Performance comparison of the native intset benchmark (left), the intset TMUNIT specification using a generated automaton (center) and using an interpreted automaton (right). The commit rate is shown above and the abort rate below.

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 [6]. 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. In [6] (Fig. 9, center), we had observed that an extra lock shift of 2 was an optimal value for workloads with sequential accesses and this is indeed the value used by SwissTM.

To observe the influence of having lock protect multiple consecutive addresses (i.e., false-sharing as mentioned in Section 4.2), we have created a workload in which threads can only conflict if there is false sharing. In Figure 7 (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 RSTM are subject to false sharing, while other implementations are not.

5.2 TMUNIT vs. Hand-Written 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 interpreted and generated techniques against the results of the existing benchmark written in C code "by hand", denoted by native.

Figure 8 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) similar to the results of the hand-written benchmark, while the interpreted version has significantly lower throughout (by a factor 3). This clearly motivates the need for the generated automaton. A second observation is that, although the interpreted version has a lower absolute throughput, it has similar relative performance (scalability).

We notice some variation in the throughput for higher number of threads. ETL performs best, followed by CTL and SwissTM. SwissTM is slower than TinySTM on this benchmark but we have just received an improved version from the authors that is expected to perform better. RSTM executes slower than other STMs again due to libstdc++ library scalability 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 4.2. As expected, the absolute throughput of the interpreted automaton is lower than for the generated automaton (and the native execution).

6. Conclusion

We have presented TMUNIT, a tool for generating workloads and testing transactional memories. TMUNIT is efficient, thanks to its automated code generation tool, and provides a workload and test specification language that is simple and powerful. TMUNIT is available online and comes with a test-suite. We hope that other researchers will contribute in extending this test-suite that use our simple domain-specific language.

Using TMUNIT, we have performed extensive experiments on five existing word-based TMs. It outlined TM specific behavior like the opacity violation of WSTM or the critical workload under which a TM behaves slowly. Our main conclusion is that, although the throughput is close to maximal under a specific workload, it may fail dramatically under a different workload. Semantic tests outlined the cause for such performance variations and indicate the need to test TM depending on the targeted application.

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. Furthermore, TM bug reports could be described and made reproducible at the TM developer site with the help of TMUNIT. Future work includes the extension of the code generation to Java for supporting object-based transactional accesses and generating critical semantic tests that are tolerated by the Java memory model.

Acknowledgements

We are grateful to Michael Spear who pointed out some causes for the obtained results, Patrick Marlier for his help on adapting WSTM for TMUNIT, and Aleksandar Dragojević for providing us with early SwissTM releases. This work is supported in part by the European Commission FP7 Velox project and the Swiss National Foundation under grant 200021-118043.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *SIGPLAN Not.*, 43(1):63–74, 2008.

[2] Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++, 2007.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[4] Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. Stretching transactional memory. Technical Report LPD-REPORT-2008-013, EPFL LPD, 2008.

[5] R. E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[6] Pascal Felber, Christof Fetzer, and Torval Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[7] Vincent Gramoli, Derin Harmanci, and Pascal Felber. Toward a theory of input acceptance for transactional memories. In *Proc. of the 12th International Conference On Principles Of Distributed Systems*, volume 5401 of *LNCS*, pages 527–533. Springer-Verlag, 2008.

[8] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[9] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.

[10] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[12] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[13] Yossi Lev and Mark Moir. Debugging with transactional memory. In *TRANSACT'06*. ACM, 2006.

[14] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *Proc. of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143. ACM, 2006.

[15] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java STM. In *Proc. of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–325, 2008.

[16] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE International Symposium on Workload Characterization*, Sept. 2008.

[17] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The limits of software transactional memory (stm): dissecting haskell stm applications on a many-core environment. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 67–78, New York, NY, USA, May 2008. ACM.

[18] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory. Technical Report RR-I-07-02.1, Université de Neuchâtel, Switzerland, February 2007.

[19] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proc. of the 20th International Symposium on Distributed Computing*, pages 284–298, September 2006.

[20] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. *SIGPLAN Not.*, 42(6):78–88, 2007.

[21] Ferad Zyulkyarov, Sanja Cvijic, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. Wormbench - a configurable workload for evaluating transactional memory systems. In *Workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA)*, October 2008.