Université de Neuchâtel
Institut d'Informatique

Rapport de Recherche — RR-I-08-08.1

TMUNIT: A Transactional Memory Unit Testing and Workload Generation Tool

Derin Harmanci, Pascal Felber, Vincent Gramoli, Martin Süßkraut, Christof Fetzer

August 19, 2008

TMUNIT: A Transactional Memory Unit Testing and Workload Generation Tool

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

Martin Süßkraut Christof Fetzer
Dresden University of Technology
Germany
{martin.suesskraut,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 not only help to develop TM, but also test them and evaluate them on a wide range of workloads. In this paper, we introduce a novel tool, TMUNIT, that is both a unit testing and workload generation tool. Its primary objective is to help TM designers specify (by example), test and optimize TM designs. TMUNIT provides a domain-specific language for describing transactional workloads for performance evaluation, and transactional schedules for unit testing. It can also automatically derive realistic workloads from (lock-based) concurrent applications to complement the small collection of existing micro- and macro-benchmarks.

1. Introduction

Transactional memory (TM) is a widely accepted parallel programming paradigm to take advantage of multi-core architectures. For the past few years, researchers from industry and academy have devoted many efforts on understanding TM behavior. Much research is focusing on finding the "right" semantics while providing good performance.

On the one hand, to express the semantics of TM and validate the behavior of a specific implementation, e.g., regarding properties like weak atomicity, one often uses code fragments that expose anomalies like non-repeatable reads *when* transactions are executed in a certain order (e.g., see [16, 1]). On the other hand, the performance of a TM implementation is evaluated with the help of micro- and macro-benchmarks.

In this paper, we propose a novel tool, TMUNIT, that can help transactional memory researchers to evaluate the semantics, validate the implementation, and improve the performance of TMs. Specifically TMUNIT targets two important goals:

- **Execution transactionalization:** applying realistic parallel executions derived from real traces (typically obtained from a lock-based concurrent application) to a specific TM that needs to be evaluated.
- **Transactional unit test:** allowing to specify reproducible interleavings of conflicting transactional operations, including pathological scenarios that rarely occur in practice, to evaluate the semantics and validate the implementation of a TM.

A difficulty when evaluating the performance of TMs is to write benchmarks that are *(i)* precise enough to emphasize TM characteristics, but also *(ii)* realistic enough to match the behavior of common applications. So far, evaluation frameworks have been dedicated to only one of these two goals [9, 12]. Some evaluate TMs at a micro level where transactions are specified as a series of read/write accesses to some data structure. Some other evaluate TMs at a macro level where a pool of threads execute a complex multi-threaded applications. On the one hand, the portion and type of operations chosen when tuning a micro-benchmark are arbitrary and might not reflect realistic settings. On the other hand, macro-benchmarks are complex to specify and represent specific and unique applications.

One of the most important questions is to what extent transactionalization can improve performance of lock-based applications. Hence, it would be ideal to be able to evaluate the performance when running an existing multi-threaded application on some transactional memories. Comparing a lock-based application to its TM counterpart may require to implement the same application by replacing all the critical sections by transactional blocks. We propose a simpler and less intrusive solution that consists in gathering traces of the original application at runtime and, once transactionalized, in using them as workload for TMs. To that end, we rely on a dynamic binary instrumentation tool, based on Valgrind [13], to identify critical sections and memory accesses at run-time. The automatic translation of this kind of trace into a TM workload is of important interest: to evaluate the benefit of transactionalization of parallel applications and to find the best suited TM configuration or implementation for a dedicated application.

A second issue in evaluating TMs is to reproduce a test that corresponds to a parallel execution, to test the correctness of an implementation (unit testing) or evaluate its semantics (e.g., whether it provides weak or strong atomicity, or if it ensures single global lock atomicity). The level of contention relies tightly on the interleaving of transactional operations and so does performance. Furthermore, pathological scenarios may appear in some—but not all—executions of the same benchmark. That is, reproducibility is a highly desirable property, especially at a micro level.

Consider, for instance, the well-known problem of dirty reads. On the left-hand side of Figure 1, transaction T_1 updates the same location x twice while transaction T_2 concurrently reads location x. Since each 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 T_2. Observe that, if

[Copyright notice will appear here once 'preprint' option is removed.]

initially x=0

T_1	T_2
x=1;	y=x;
x=2;	

Definitions: _y = 0, x = 0;
Transactions: T1 := W(x,1), @L1, W(x,2);
T2 := R(x,_y);
Schedules: T1@L1, T2, T1;
Invariants: [T2:_y != 1];

Figure 1. A simple pathological scenario that may lead to dirty read ($y = 1$) on the left-hand side, and a corresponding TMUNIT specification to test if the TM avoids the dirty read on the right-hand side. Note that $\_y$ is a thread-local variable and we define a label ($@L1$) in $T1$ to specify the interleaving of operations in the schedule.

we just require that two threads execute T_1 and T_2 concurrently on a TM, the dirty read may well not occur at runtime. To test appropriately that the TM avoids this problem, one must specify a certain interleaving of conflicting operations. On the right-hand side of Figure 1, we show the TMUNIT specification of a dirty read unit test. The interleaving of the transactions is defined in a schedule that enforces the read operation of x to occur between the two write operations of T_1. The invariant checks, in the context of T_2, whether this specific schedule leads to the dirty read problem ($y = 1$). Such a unit test language specification is of crucial importance for testing TM behaviors in specific situations, notably in scenarios with concurrent transactional and non-transactional accesses.

Contributions

TMUNIT has been designed to address our two goals of execution transactionalization and transactional unit test. First, TMUNIT can convert automatically a given multi-threaded trace into a transactional benchmark. Second, the user can write unit tests using a simple language and execute them on different word-based TM implementations.

TMUNIT provides a domain-specific language for specifying TM workloads that is both 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. Benchmarks can be executed by dynamically interpreting the workload specification and mapping transactional accesses to an underlying TM, or for best performance by generating C code to be compiled into a standalone application.

TMUNIT goes beyond existing TM benchmarks *(i)* by translating automatically multi-threaded traces into TM workloads, and *(ii)* by making it straightforward to test specific reproducible executions on different TMs.

Roadmap

In Section 2, we present the architecture of TMUNIT. and describe our domain-specific language. In Section 3, we explain how to reuse multi-threaded application traces as TM workloads. Section 4 validates our tool in terms of execution efficiency, TM performance tests, and TM unit tests. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2. TMUNIT Architecture and Language

In this section, we briefly describe the architecture components of TMUNIT and explain how they interact (see Figure 2). We then describe the domain specific language of TMUNIT.

2.1 Architecture Overview

As mentioned earlier, TMUNIT executes a workload on a dedicated TM. This workload corresponds either to a real trace taken from an existing application (see Section 3) or to a synthetic description written in a domain-specific language (see Section 2.2). TMUNIT can either execute this workload dynamically or convert it into a C program to reduce the runtime overheads. TMUNIT also records performance statistics and test results.

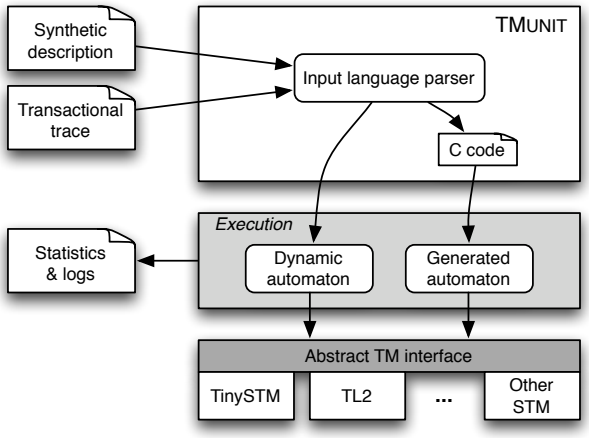

Figure 2. Architectural overview of TMUNIT.

A test is modeled as an automaton performing transactional operation calls through the underlying TM. TMUNIT can use two different automata, depending on whether the configuration file is dynamically interpreted or translated into a C program. The choice of the execution mode is done by the user. If desired, the execution of either automaton may output a trace of the execution and detailed performance statistics.

TMUNIT uses an abstract TM interface to map transactions on the underlying TM implementation. We have successfully tested it with TinySTM [7] and TL2 [5] implementations but any word-based TM can be straightforwardly supported. Note that the TM can be linked as a dynamic library and, hence, no recompilation of TMUNIT is necessary when testing a new TM.

2.2 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 benchmarking). We give a high level description of TMUNIT's generic language to specify TM workloads. Automatic workload generation is presented in Section 3.

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.

```
1 T1 := R(x), R(y), W(x), W(y);                    // R = read, W = write
2 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, $T1$, 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, $T2$ 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 predetermined interleaving of operations.
6. The *invariants* section specifies assertions that must be valid at each step of a schedule.

2.3 Unit Test Specification

A *unit test* refers to a simple 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 [6]. 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 [2].[1] The write to z by $T2$ 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 determine 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].

Operations and transactions. We assume a single address space of bounded size. Transactions can only communicate by writing to, and reading from, the shared address space. We denote reads by `R` and writes by `W` and each of these two operations needs to be applied to a shared memory variable.

Variables are defined in the *definitions* section and are either word-size integers or arrays of word-size integers. Undeclared variables are assumed to be word-size 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 are 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.

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. We permit to optionally record the result 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 a 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, parentheses, and yielding 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, $T1$ contains two operations (Line 5) while $T2$ contains one operation and an *if* statement (Line 6). Label `@L1` is used in $T1$'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 at run-time 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 anytime during the execution, then the test fails. The program prints an error message and optionally stops.

We need to be more precise about the semantics of invariants. In particular, we need to define the evaluation context of variables within boolean expressions used in assertions and invariants.

- Local variables are evaluated in the context of the transaction they are used in. For example, local variable $\_y$ in expression `[T2:_y != 1]` (Figure 1) is evaluated in the context of transaction $T2$. If $T2$ has not yet been executed, $\_y$ evaluates to its initial value and otherwise, $\_y$ evaluates to its most recent value assigned within $T2$. Note that the value of a local variable is hence always well defined.
- Unprotected accesses, like `[z != 1]` (Line 12 of Listing 2), are evaluated by directly reading the memory location at which z is stored, i.e., without using the underlying TM.

[1] 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.

- Protected accesses, like `[R(z) != 1]` are evaluated in the context of an anonymous transaction, i.e., this expression returns the most recently committed value of shared variable z.
- Protected accesses within the context of a given transactions, like `[T1:R(x) != 1]`, see also uncommitted variables that might be rolled back later if the transaction aborts. This invariant is only evaluated if the transaction is currently active.

As an example, the schedule presented at Line 9 of Listing 2 expresses a "zombie transactions" scenario [2] where an intermediate dirty read may occur. Transaction $T1$ is executed up to label $L1$, thus, only the first of its two write operations is executed. Then, transaction $T2$ is executed before $T1$ resumes. As a result, this schedule forces $T2$ to read x between the two writes of $T1$. If $T2$ reads the 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 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).

```
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   T_update:_f = 0;                                   // flag to alternate between inserts and removes
11   T_update:_v = 0;                                   // last value added
12
13 Transactions:                                        // specification of transactions
14   T_search := {# k = [0 .. NB−1] : R(m[2∗k]), R(m[2∗k+1]) },    // search element
15               R(m[2∗NB]) ;
16   T_update := {? [_f == 0] :
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]) },    // del. 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   P_1, P_2 := < T_search : 80% | T_update : 20% >∗;
```

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

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 nondeterministically 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_search` 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 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_update` 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 why 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 prevent 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 4, 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 essentially 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_search` with probability $\frac{4}{5}$ and transaction `T_update` with probability $\frac{1}{5}$ in an endless loop. Such experiments are interrupted after a specified timeout or with a signal.

The language also offers several other features, like the ability to insert explicit delays (`D`) between operations and between transactions, explicit jumps (`J`) to labels, fine control over mapping of variables to shared memory, etc.

3. Trace Generation

TMUNIT supports execution transactionalization, i.e., it helps to transform existing lock-based applications into workloads for TM. Our aim is to create more realistic workloads for TMs—in comparison to the current state of the art. One could argue that future "pure" TM-based applications will have different performance characteristics than lock-based applications. For example, the average length of transactions of TM-based applications might be longer than the average length of critical sections in lock-based applications. However, we expect that, at least initially, existing lock-based applications will be transformed manually or automatically (e.g., using lock elision approach [19, 14]) into TM-based applications. Hence, it is a worthwhile goal to provide workloads derived from real concurrent lock-based applications.

Manual transformation of an existing lock-based application typically requires quite some effort. While automatic transformations have already been proposed [19], this is difficult to achieve in unmanaged languages like C and C++. One major advantage of using TMUNIT for TM workloads is that we can abstract away from many of these problems since we are only interested in emulating the access pattern to the TM and not in providing any application semantics.

Our approach in generating a workload from an existing lock-based application is as follows. We dynamically instrument the application binary such that we detect when a thread acquires and releases a lock. Within such a critical section, we log all memory accesses performed by a thread. Our implementation performs this logging with the help of a Valgrind [13] *skin*. Valgrind is a dynamic binary instrumentation tool and a skin is an application-independent script that defines how programs should be instrumented.

A compiler optimized for TM can use static analysis to distinguish between accesses to local and shared variables within transactions: only accesses to shared variables need to be performed with the help of a TM. The rollback of local variables could then be performed with a lower-cost mechanism. Hence, we process the generated log to detect which memory locations are accessed by more than one thread, i.e., which memory locations are shared. To reduce the size of the logs, we do not record stack accesses in our log because they are most likely not shared. In Section 4.3, we show measurements for the number of memory accesses and the percentage of shared memory accesses for several applications.

The set of memory locations accessed within a critical region typically varies during an execution. For example, consider a critical section protecting the consistency of a collection to which we add and remove objects. Our experiments show that defining individual TMUNIT transactions for all operations on this set is often not feasible. We instead compress this set as follows (see Figure 3). We first renumber all addresses such that they become an index in some dense array (in Figure 3 it corresponds to array m). Second, we record the sequencing of array accesses of a critical section in a weighted directed graph. The root node of the graph represents the start of the critical section and its end is represented by a terminal node. All other nodes represent either a read or write access to an entry in the array. The outgoing edges define all potential successor nodes. The weights of edges determine how often we saw a given successor node during the trace generation. By construction, the terminal node can be reached from each node in the graph.

The graph is used by TMUNIT to generate dynamically the sequence of read and write operations of the transaction that represents the critical section. Starting at the root, TMUNIT follows a random path through the graph. Say, we are currently at node N, which has a set of direct successors $succ(N)$. The probability to follow a given edge (N, S) (where $S \in succ(N)$) is determined by the weight of edge (N, S) divided by the sum of the weight of all outgoing edges of N. The random path is terminated when reaching the terminal node.

For some applications, the graph representing a critical section is still too large to fit in memory. For such cases, we reduce its size during the post processing of the traces to approximately K nodes, where K is a constant (given as a command line argument). To prune the graph, we generate offline a number of random paths (as described above) and we mark each node on such a path. Initially, all nodes except the root and terminal nodes are unmarked. We generate random paths until at least K nodes are marked. The graph used by TMUNIT consists of all marked nodes and all edges between marked nodes. Note that for this new graph the terminal node can still be reached from each node in the graph. As before, TMUNIT will use this graph to generate the read and write operations of the transaction that represents the critical region.

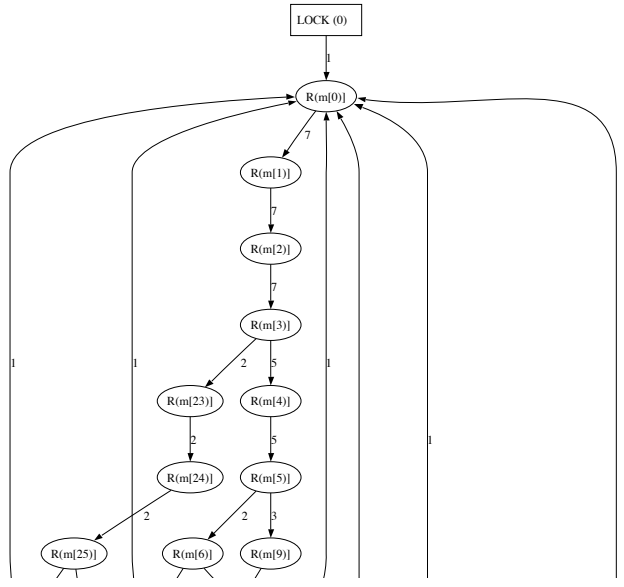

Figure 3. Extract of read/write graph for a tiny instance of the vacation benchmark: vacation arguments -c4 -n4 -q60 -u90 -r8 -t2.

4. TMUNIT Experimentations

In this section, we reproduce existing results using our tool to validate it, we evaluate the overhead induced by our intermediary domain-specific language, and we run some experiments on TM implementations to test their performance.

4.1 Experimental setup

All tests were run on an 8-core Intel Xeon (X5365) machine at 3 GHz running Linux 2.6.24-19-generic (64-bit). We used version 0.9.5 of the x86 port of TL2 [5], version 0.9.5 of TinySTM [7] and version 0.9.9 of STAMP [12]. All tests were run in 64-bit mode and were compiled using gcc 4.2.3.

We have experimented with two different TinySTM design: encounter-time locking (ETL), which acquires locks as shared memory locations are being written; and commit-time locking (CTL), which only acquires locks when the transaction tries to commit. Unlike the algorithm of TL2, the CTL variant of TinySTM supports incremental snapshot extension (as in [15]), which improves performance on some workloads.

4.2 Performance testing with synthetic specifications

Our first aim is to validate the standalone code generation by comparing the performance obtained with TMUNIT using both the interpreted and compiled automata, against the performance of the equivalent application hand-written in C.

Integer set linked list benchmark. The integer set linked list benchmark is one of the most widely used benchmarks for evaluating performance of TM implementations. It consists of a single shared data structure, a sorted linked list, in which integer values can be concurrently inserted, removed, or searched. 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, as in [7]) according to a given probability distribution.

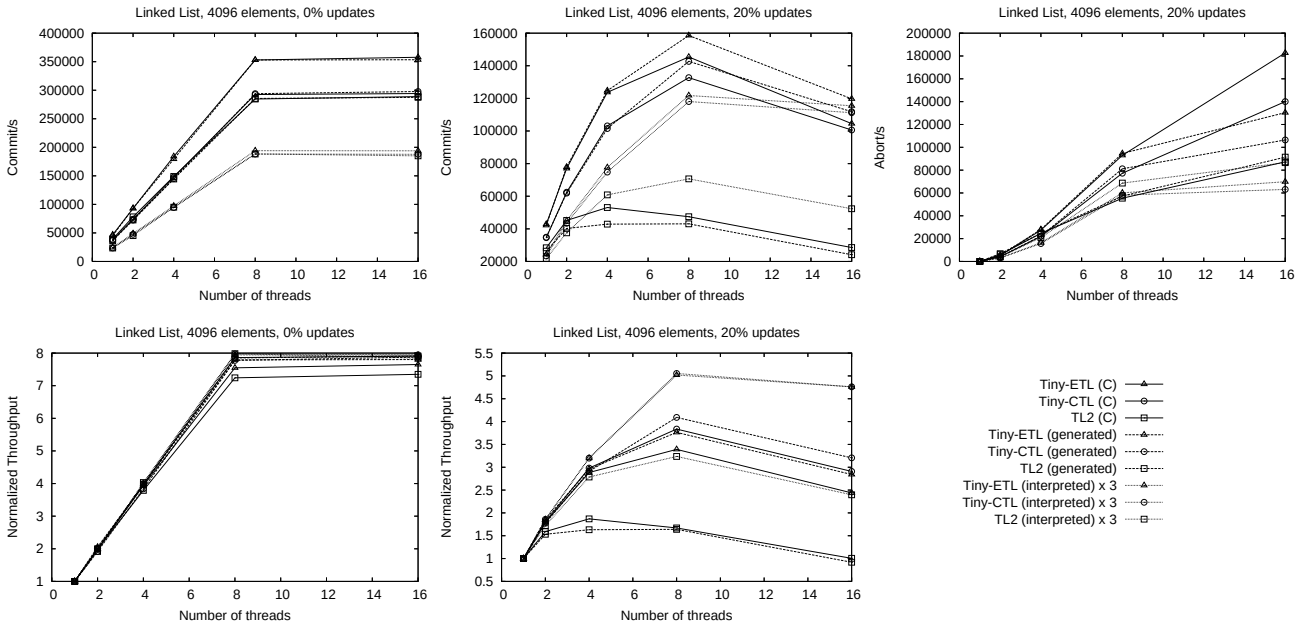

Figure 4. Comparison of the performance of hand-written C code and TMUNIT for the linked list benchmark (the list initially contains 4096 elements).

TMUNIT *vs. hand-written C.* To verify that performance tests of TMUNIT provide meaningful results, we have compared the results obtained by running the linked list TMUNIT specification as it appears in Listing 3 against hand-written C code. Tests were performed with two distributions of update transactions: 0% and 20%. Results are shown in Figure 4. Note that in the Figure the performance results for the interpreted mode of TMUNIT are multiplied by a factor of 3 to make it easier to compare the shapes of the data series. This factor accounts for the overhead of the dynamic execution of the automaton.

In the case of read only transactions (top-left graph), we observe that the throughput of the native application is virtually identical to that of the TMUNIT generated automaton. The interpreted automaton has lower absolute throughput but similar relative performance (scalability). This is confirmed by observing the normalized throughput function of the single-thread performance (bottom-left graph). One can note that all TM designs provide similar performance for the read-only workload.

When introducing 20% update transactions, we notice in the commit throughputs (top-center graph) of the generated automaton and the native application are highly similar for the different designs, with some variation for higher number of threads. ETL performs best, followed by CTL and TL2. The reason why TL2's performance is lower than TinySTM can be explained by the differences in timestamp management:[2] the latter performs snapshot extension, which is especially useful with large read sets as in the linked list benchmark. The absolute throughput of the interpreted automaton is lower as expected. The number of aborts (top-right graph) provides also consistent results for all the execution modes and TM implementations.

When comparing the normalized throughput (bottom-center graph), one notices that results for the generated automaton and the native application are similar, but they differ for the interpreted automaton that shows better scalability. This can be explained by the fact that, as the tests run approximately 3 times slower, contention is slightly reduced (this can be observed in the number of aborts) and hence relative performance increases with the number of threads. One should note, however, that the general shape of the scalability curves remains consistent.

[2] Thanks to Michael Spear who pointed this out to us.

4.3 Performance testing with traces

We evaluated our workload generator with traces from the vacation STAMP [12] benchmark (using Vacation, k-Means and Genome) and three lock-based applications: Audacity 1.3.2-beta [3] (graphical sound edition tool), Blender 2.46 [4] (3D modeler and raytracer), and GIMP 2.4.6 [17] (image manipulation program). The main difficulty is the size of the traces. Figure 5 shows the number of memory accesses we have recorded for our evaluated applications (full trace). The number of accesses ranges from 17 millions for the k-Means STAMP benchmark to over 1.5 billions for the vacation STAMP benchmark.

Figure 6 shows the workload characteristics we extracted from the traces. *Shared* accesses read and write memory addresses touched by more than one thread. *Read only and thread local* accesses read memory addresses that were accessed by only one thread and never written. *Read only* accesses read memory addresses that were never written in the whole trace (excluding *thread local* accesses). *Thread local* accesses read and write memory addresses that were only touched by one thread (excluding *read only* accesses). The thread local accesses do not include stack accesses because they are excluded at trace recording time.

The workload of the lock-based applications is dominated by thread local accesses. Applications often allocate memory on the heap intended to be used as thread local storage. Read only accesses read for instance global constants. The workload of STAMP benchmarks is dominated by accesses to read only memory (vacation benchmark) or accesses to shared memory (k-Means and Genome).

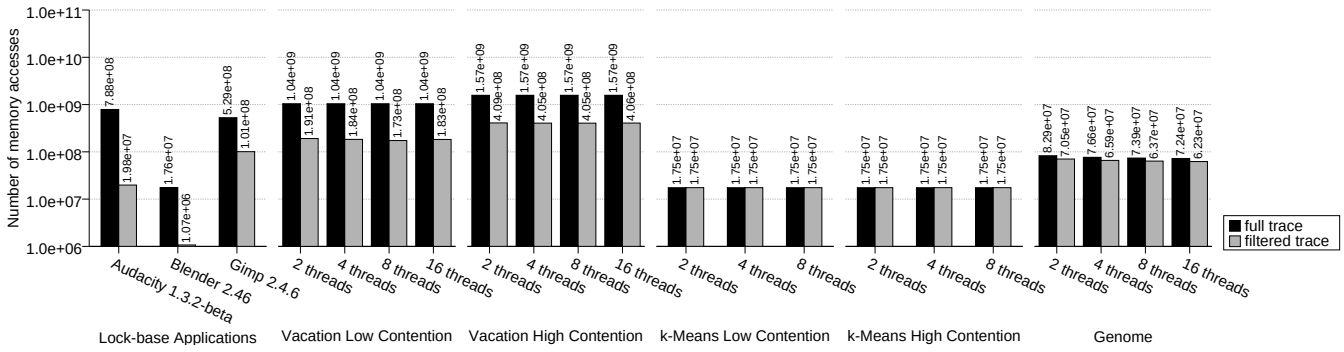

Figure 5. Size of full and filtered traces in number of memory accesses.

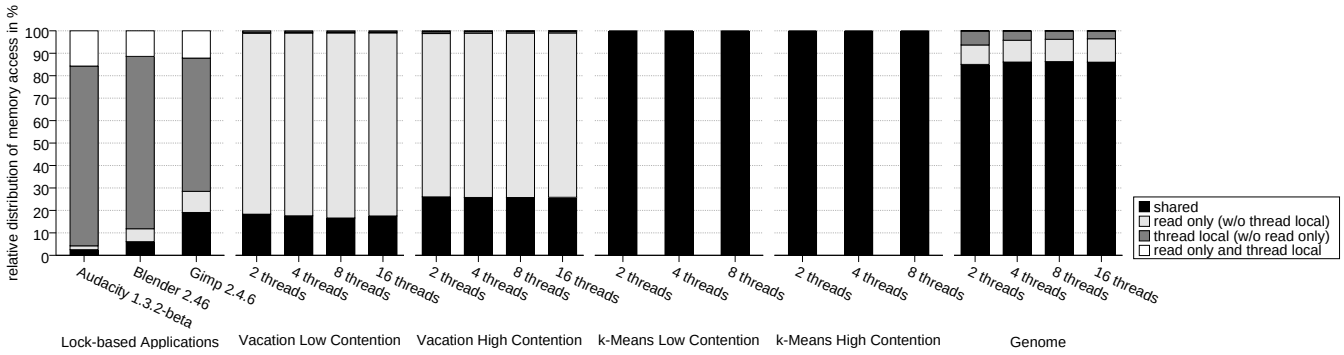

Figure 6. Characteristics of recorded workload.

STAMP traces. Our approach for tracing STAMP benchmark applications differs from the one described in Section 3. STAMP is written for benchmarking TM implementations. We wrote a *tracer STM* for STAMP. The tracer STM records every read and write access and writes them to the trace together with the memory address accessed and the current thread identifier. Additionally, the tracer STM writes transaction start and commit to the trace. A transaction is identified by the current value of the instruction pointer at the time it starts. Transactions are serialized with a simple global lock.

The usage of the tracer STM also explains why the traces do not contain as many thread local accesses as for the lock-based applications. Only memory accesses carefully annotated by the STAMP authors are visible to the benchmarked TM implementation.

Figure 7 compares the native execution of the vacation benchmark with TMUNIT running workload derived from vacation traces. The differences between the native executions and the traces driven are due to filtering of the traces and sampling of the models. The models do not yet account for the time a thread spends executing non-transactional code, i.e., the traces only reproduce transactional access and hence, permit much higher commit rates. On the other hand, TMUNIT consistently reproduces the relative performance of the native executions. That shows TMUNIT can be used to compare the performance of TM implementations by replaying recorded traces.

Traces from lock-based applications. We recorded traces of three popular lock-based applications by instrumenting all memory accesses (see Section 3). Traces were recorded while a user exercised the applications (e.g., by a user editing audio with Audacity or rendering an image with Blender). Note that a great advantage of using traces to generate TM benchmarks is that we can generate benchmarks from interactive applications. Exercising such applications within a benchmark script in not easy.

Using the generated traces directly as input for TMUNIT is infeasible due to their size (see Figure 5). The overhead of holding the traces in memory or on disc would dominate the performance measurements. Also, when generating C code from these traces, we experienced some compiler limitations. Notably, processing source code files of hundreds of megabytes often leads the compiler to abort or to run for several hours.

To address the problem of large model sizes, we implemented the sampling approach described in Section 3. In Figure 8 we show the reduction of the models for several applications in terms of number of nodes. The models for Audacity and k-Means were small enough to be used without sampling.

Figure 9 shows the commit rate (left) and the performance of TL2 and four variants of TinySTM for the applications. These variants are commit-time locking (CTL), encouter-time locking (ETL) optionally with contention manager (ETL_CM) and write-through (WT) policy. We can see that are orders of magnitude difference in the throughput of the traces. The right graph of Figure 9 shows the speedup of the variants of TinySTM for the different applications with respect to TL2. We observe a substantial speedup of TinySTM over TL2.

4.4 Using TMUNIT to identify liveness problems

In this section, we demonstrate how TMUNIT can effectively be used for testing and evaluation purposes. We present an example that exhibits liveness problems, not much different from a livelock,

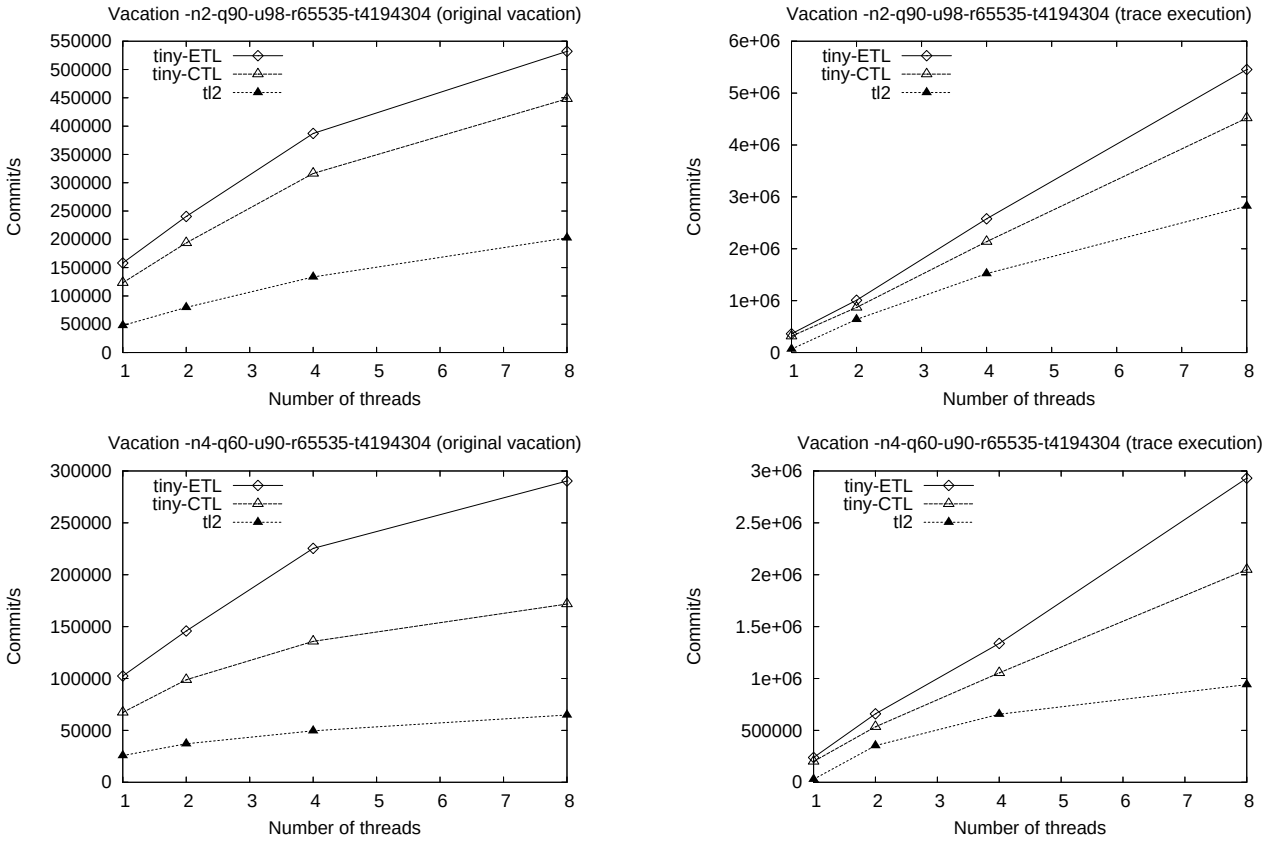

Figure 7. The commit rate per second for the vacation benchmark comparing native execution (left) with trace driven using TMUNIT (right). The upper half shows a low contention setup, the lower half a high contention setup.

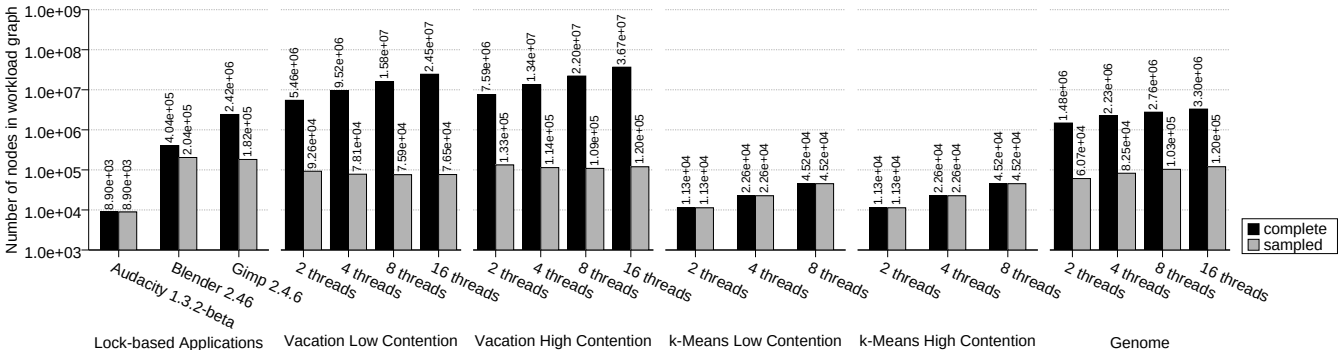

Figure 8. Comparison of complete and sampled models.

where some transactions are repeatedly aborted by others and cannot progress. This illustrates how one can simply generate pathological or extreme scenarios to compare TM designs.

We use a classical "bank" workload where one thread computes the sum of the balances of all accounts (long read-only transaction) while the other threads concurrently perform transfers (short update transactions). TM designs with invisible reads (like TL2 and TinySTM) and no fair contention management strategy are expected to suffer from a lack progress for the long read-only transactions as conflicting updates from concurrent transactions will lead to failed validation at commit time. The problem is illustrated in Figure 10 (left), where conflicting writes by 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.

The objective of this test is to compare the behavior, in terms of progress, of TM designs with and without contention management. To that end, we focus on the thread that executes long read-only transactions and we observe its throughput for various configurations. Contention management is implemented by a fair contention

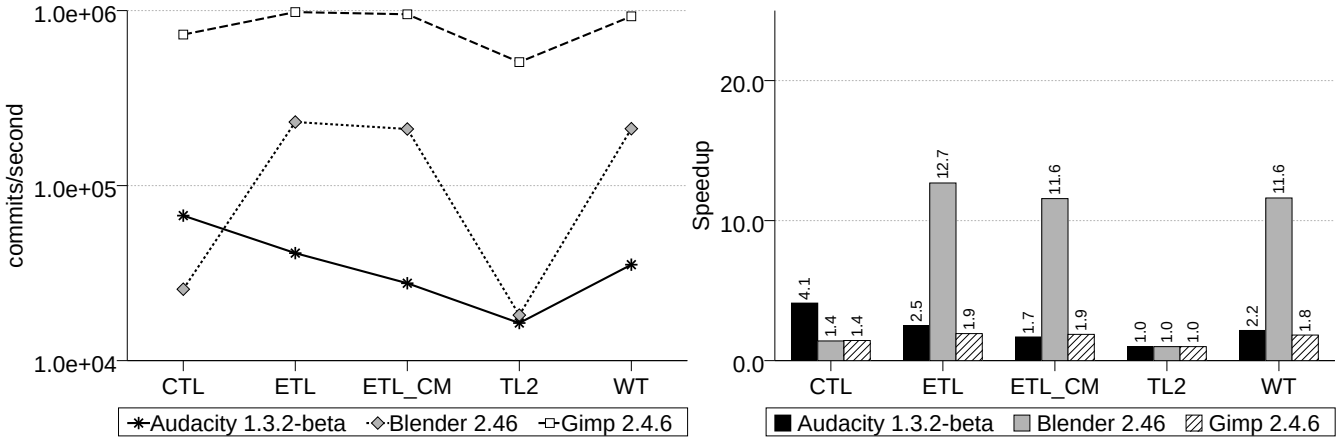

Figure 9. The commit rate per second of the traces using different STM variants (left) and the speedup of various TinySTM variants versus TL2 (right).

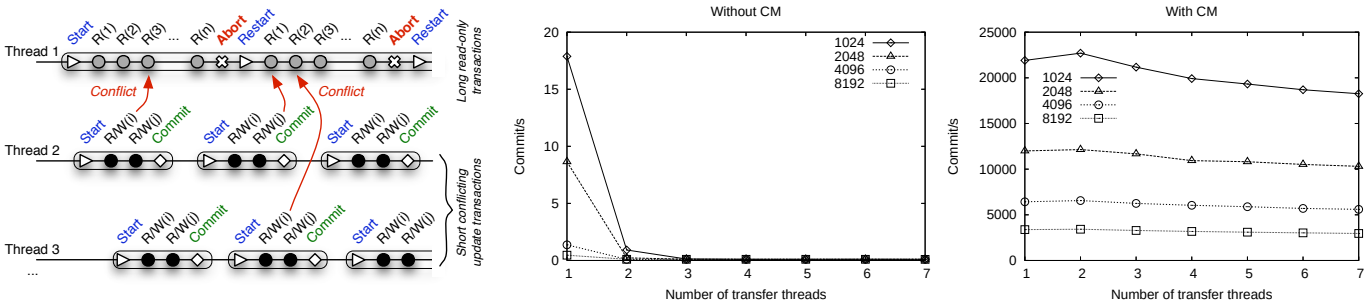

Figure 10. Illustration of the lack of progress of long read-only transactions in the bank benchmark (left). Comparison of the commit rate without (center) and with (right) contention manager for various numbers of bank accounts and threads performing transfers.

manager that associates priority with threads. Upon conflict, the highest priority transaction can proceed while the other one must abort. Upon restart, transactions increase their priority. The contention manager also switches to visible reads if it fails validation (because of a conflicting write on a read memory location). Using this workload, one can evaluate whether the contention manager guarantees progress.

```
1  Definitions:                                   // variables and constants
2    NB = 8192;                                   // number of accounts (constant)
3    a[1 .. NB] = 0;                              // memory range for accounts
4    SRC = <1 .. NB>;                             // random value (constant) in range 1...NB
5    DST = <1 .. NB>;                             // random value (constant) in range 1...NB
6
7  Transactions:                                  // specification of transactions
8    T_transfer := R(a[SRC]), R(a[DST]), W(a[SRC]), W(a[DST]) ;   // transfer
9    T_balance := {# k = [1 .. NB] : R(a[k]) } ;                 // compute balance
10
11 Threads:                                       // specification of threads
12   P_1 := T_balance*;
13   P_2, P_3, P_4, P_5, P_6, P_7, P_8 := T_transfer*;
```

Listing 4. Specification of the bank benchmark that exhibits lack of progress (8 threads).

Listing 4 shows the TMUNIT specification for the bank benchmark. One can appreciate the simplicity of this description as compared to the effort necessary for writing a standalone benchmark. Note that, to exacerbate the lack of progress problem even more, one could add delays in `T_balance` to artificially slow it down and further increase the likelihood of conflicts.

As we can observe in Figure 10 (center), the commit throughput without contention manager drops to 0 with as little as three threads performing transfers, and it only reaches 18 with a single update thread. To better isolate the cause of such behavior, TMUNIT can output detailed execution traces that can be inspected a posteriori. When using a contention manager, we observe in Figure 10 (right) that the commit throughput is almost constant independent of the number of update threads, which demonstrates that it provides fairness among competing transactions.

5. Related Work

Testing. Two types of testing tools for TMs exist in the literature. The first type of tools (e.g., [11]) are rather used for verification purpose as they test a series of schedules in a row. Such tools do not support unit tests. The second type of testing tools (e.g., [10]) are dedicated to debugging of TMs and do not produce specific schedules. The direct application of such tools is step-by-step debugging, a topic out of the scope of our paper.

In [8], the input acceptance of several TMs were tested with specific transaction patterns, however, the interleaving of operations in each workload could not be predetermined as it is in TMUNIT.

Benchmarking. Several frameworks have already been proposed in the literature to evaluate performance of TMs. STMBench7 [9] provides the user with a complex data structure and update/traversal primitives on this data structure. These primitives are made tunable so that the user may indicate, for example, whether execution should be dominated by write operations or by read operations. STAMP [12] presents a list of several real-world applications used as benchmarks. While STAMP provides an invaluable tool for TM developers, it is limited by the number and type of benchmarks available: extending this benchmark suite requires to fully implement new applications. SPLASH-2 [18] is another benchmarking framework, dedicated however to study shared-memory management in parallel applications but not the performance of transactional memory.

As far as we know, there is no existing tool that can automatically extract traces from concurrent applications to be used as workload for STM.

6. Conclusion

We have presented TMUNIT, a tool for generating workloads and testing transactional memories. TMUNIT provides a workload and test specification language that is both simple to use and powerful. It is efficient, thanks to its automated code generation tool. Finally, TMUNIT can automatically translate lock-based parallel applications into transactional benchmarks.

We have preformed extensive experiments with TMUNIT on distinct TMs to illustrate the importance of unit testing and benchmarking. For instance, the linked list benchmarking of TinySTM indicate that encounter-time locking supersedes commit-time locking under various circumstances. The benchmarking of the various application workloads illustrate the transactionalization of lock-based applications. The livelock unit tests motivate the use of contention managers.

We believe that TMUNIT is an important tool for the research community on transactional memory since it helps TM designers in verifying the behavior and performance of their TM, and may outline potential issues on dedicated tests.

When TM-based applications become more prevalent, we believe that TMUNIT can have several other uses. For example, one could specify the required TM semantics of an application with the help of TMUNIT: 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 of a TM could be described and made reproducible at the TM developer site with the help of TMUNIT. Last but not least, we plan to evolve TMUNIT such that it becomes a comprehensive unit testing and benchmarking tool for TM-based applications.

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] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[3] Audacity, 2008. `http://audacity.sourceforge.net/`.

[4] Blender Foundation, 2008. `http://www.blender.org/`.

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

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

[7] Pascal Felber, Christof Fetzer, and Torval Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008.

[8] Vincent Gramoli, Derin Harmanci, and Pascal Felber. Toward a theory of input acceptance for transactional memories. Technical Report LPD-REPORT-2008-009, Distributed Programming Laboratory - EPFL, May 2008.

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

[10] Yossi Lev and Mark Moir. Debugging with transactional memory. In *TRANSACT '06: Proceedings of the 1st ACM SIGPLAN Workshop on Transactional Computing*. ACM, 2006.

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

[12] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, Sept. 2008.

[13] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[14] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

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

[16] 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.

[17] The GIMP Team, 2008. `http://www.gimp.org/`.

[18] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[19] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 129–154. Springer, 2008.