Stretching Transactional Memory

Aleksandar Dragojević Rachid Guerraoui Michał Kapałka
Ecole Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, I&C, Switzerland
{aleksandar.dragojevic, rachid.guerraoui, michal.kapalka}@epfl.ch

Abstract

Transactional memory (TM) is an appealing abstraction for programming multi-core systems. Potential target applications for TM, such as business software and video games, are likely to involve complex data structures and large transactions, requiring specific software solutions (STM). So far, however, STMs have been mainly evaluated and optimized for smaller scale benchmarks.

We revisit the main STM design choices from the perspective of complex workloads and propose a new STM, which we call SwissTM. In short, SwissTM is lock- and word-based and uses (1) optimistic (commit-time) conflict detection for read/write conflicts and pessimistic (encounter-time) conflict detection for write/write conflicts, as well as (2) a new two-phase contention manager that ensures the progress of long transactions while inducing no overhead on short ones. SwissTM outperforms state-of-the-art STM implementations, namely RSTM, TL2, and TinySTM, in our experiments on STMBench7, STAMP, Lee-TM and red-black tree benchmarks.

Beyond SwissTM, we present the most complete evaluation to date of the individual impact of various STM design choices on the ability to support the mixed workloads of large applications.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.8 [*Software Engineering*]: Metrics—performance measures

General Terms Measurement, Performance, Experimentation

Keywords Software transactional memories, Benchmarks

1. Introduction

Transactional memory (TM) is an appealing abstraction for making concurrent programming accessible to a wide community of non-expert programmers while avoiding the pitfalls of critical sections. With a TM, application threads communicate by executing operations on shared data inside lightweight in-memory *transactions*. A transaction performs a number of actions and then either *commits*, in which case all the actions are applied to shared data *atomically*, or *aborts*, in which case the effects of those actions are rolled back and never visible to other transactions. From a programmer's perspective, the TM paradigm is very promising as it promotes program composition [20], in contrast to explicit locking, while still providing the illusion that all shared objects are protected by some global lock. Yet, it offers the possibility of performance comparable to hand-crafted, fine-grained locking.

A possible target of TMs are large applications such as business software or video games: the size of these applications make them ideal candidates to benefit from emerging multi-core architectures. Such applications typically involve dynamic and non-uniform data structures consisting of many objects of various complexity. For example, a video gameplay simulation can use up to 10,000 active interacting game objects, each having mutable state, being updated 30–60 times per second, and causing changes to 5–10 other objects on every update [40]. Unless a TM is used, making such code thread-safe and scalable on multi-cores is a daunting task [40]. The big size and complexity of such applications can, in turn, easily lead to large transactions, for these can naturally be composed [20]. Some TM interfaces [1], in fact, promote the encapsulation of entire applications within very few transactions.

The motivation of this work is to explore the ability of software mechanisms to effectively support mixed workloads consisting of small and large transactions, as well as possibly complex data structures. We believe this to be of practical relevance because even if hardware TM support becomes widely available in the future, it is likely that only smaller-scale transactional workloads will be fully executed in hardware, while software support will still be needed for transactions with large read and write sets. For example, the hybrid hardware/software scheme proposed in [26] switches from full hardware TM to full software TM when it encounters large transactions. The ability of STM systems to effectively deal with large transactions will be crucial in these settings as well.

Since the seminal paper on a *software* TM (STM) that supported *dynamic* data structures and *unbounded* transactions [22], all modern STMs are supposed to handle complex workloads [22, 27, 10, 31, 21, 2, 35, 29]. A wide variety of STM techniques, mainly inspired by database algorithms, have been explored. The big challenge facing STM researchers is to determine the right combination of strategies that suit the requirements of concurrent applications—requirements that are significantly different than those of database applications. So far, however, most STM experiments have been performed using benchmarks characterized by small transactions, simple and uniform data structures, or regular data access patterns. While such experiments reveal performance differences between STM implementations, they are not fully representative of complex workloads that STMs are likely to get exposed to once used in real applications. Worse, they can mislead STM implementors by promoting certain strategies that may perform well in small-scale applications but are counter-productive with complex workloads. Examples of such strategies, which we discuss in more details later in the paper, include the following.

1. The *commit-time locking* scheme, used for instance in TL2 [10], is indeed effective for short transactions, but might waste significant work of longer transactions that eventually abort due

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06...$5.00

to write/write conflicts. This is because write/write conflicts, which usually lead to transaction aborts[1], are detected too late.

2. The *encounter-time locking* scheme, used by most STMs, e.g., TinySTM [31], McRT-STM [35, 29], and Bartok-STM [21] immediately aborts a transaction that tries to read a memory location locked by another transaction. Hence, read/write conflicts, which can often be handled without aborts, are detected very early and resolved by aborting readers. Long transactions that write memory locations commonly read by other transactions might thus end up blocking many other transactions, and for a long time, thus slowing down the system overall.
3. The *timid contention management* scheme, used by many STMs, especially word-based ones such as TL2 and TinySTM, and which aborts transactions immediately upon a conflict, favors short transactions. Contention managers such as Greedy [16] or Serializer [34] are more appropriate for large transactions, but are hardly ever used due to the overhead they impose on short transactions.

It is appealing but challenging to come up with strategies that account both for long transactions and complex workloads, as well as for short transactions and simple data structures: these might indeed typically co-exist in real applications. This paper is a first step towards taking that challenge up. We perform that step through SwissTM, a new lock- and word-based STM. The main distinctive features of SwissTM are:

- A conflict detection scheme that detects (a) write/write conflicts eagerly, in order to prevent transactions that are doomed to abort from running and wasting resources, and (b) read/write conflicts late, in order to optimistically allow more parallelism between transactions. In short, transactions eagerly acquire objects for writing, which helps detect write/write conflicts as soon as they appear. This also avoids wasting work of transactions that are already doomed to abort after a write/write conflict. By using invisible reads and allowing transactions to read objects acquired for writing, SwissTM detects read/write conflicts late, thus increasing inter-transaction parallelism. A time-based scheme [10, 33] is used to reduce the cost of transaction validation with invisible reads.
- A two-phase contention manager that incurs no overhead on read-only and short read-write transactions while favoring the progress of transactions that have performed a significant number of updates. Basically, transactions that are short or read-only use the simple but inexpensive timid contention management scheme, aborting on first encountered conflict. Transactions that are more complex switch dynamically to the Greedy mechanism that involves more overhead but favors these transactions, preventing starvation. Additionally, transactions that abort due to write/write conflicts back-off for a period proportional to the number of their successive aborts, hence reducing contention on memory hot spots.

We evaluate SwissTM with state-of-the-art STMs by using benchmarks that cover a large part of the complexity space. We start with STMBench7 [18], which involves (1) non-uniform data structures of significant size, and (2) a mix of operations of various length and data access patterns. Then, we move to Lee-TM [4]—a benchmark with large but regular transactions—and STAMP [8]—a collection of realistic medium-scale workloads. Finally, we evaluate SwissTM with a red-black tree microbenchmark that involves very short and simple transactions. SwissTM outperforms state-of-the-art STMs—RSTM [27], TL2 [10], and TinySTM [31]—in all the considered benchmarks. For example, in the read-dominated workload of STMBench7 (90% of read-only operations), SwissTM outperforms the other STMs by up to 65%, and in the write-dominated workload (10% of read-only operations)—by up to 10%. Also, SwissTM provides a better scalability than the other STMs, especially for read-dominated and read-write (60% of read-only operations) workloads of STMBench7.

[1] Pure write/write conflicts do not necessarily lead to transaction aborts, but are very rare—most transactions read memory locations before updating them.

STM design choices			
Acquire	*Reads*	*CM*	*Effectiveness*
lazy	invisible	any	+
eager	visible	any	+
eager	invisible	Polka	+
eager	invisible	timid or Greedy	++
mixed	invisible	timid or Greedy	+++
mixed	invisible	2-phase	++++

Table 1. A summary comparison of the effectiveness of selected combinations of STM design choices in mixed workloads.

We compare SwissTM to RSTM, TL2, and TinySTM for two reasons.

- They constitute the state-of-the-art performance-wise, among the publicly available library-based STMs. Furthermore, just like SwissTM, they can be used to manually instrument concurrent applications with transactional accesses. Indeed, our goal is to evaluate the performance of the core STM algorithm, not the efficiency of the higher layers such as STM compilers. We did not use for instance McRT-STM [35, 29], because it does not expose such a low-level API to a programmer. Evaluating STM-aware compilers (which naturally introduce additional overheads above the low-level STM interface [42, 6]) is largely an orthogonal issue;
- They represent a wide spectrum of known TM design choices: *obstruction-free* vs. *lock-based* implementation, *eager* vs. *lazy* updates, *invisible* vs. *visible* reads, and *word*-level vs. *object*-level access granularity. They also allow for experiments with a variety of contention management strategies, from simply aborting a transaction on a conflict, through exponential back-off, up to advanced contention managers like Greedy [16], Serializer [34], or Polka [41].

We report on our SwissTM (trial-and-error) experience, which we believe is interesting in its own right. It is the first to date that evaluates the ability of software solutions to provide good performance to large transactions and complex objects without introducing significant overheads on short transactions and simple data structures. We evaluate the individual impact of various STM design choices on the ability to support mixed workloads. A summary of our observations, is presented in Table 1.

From an implementation perspective, we also evaluate the impact of the locking granularity. Word-based STM implementations used so far either word-level locking (e.g., TL2 and TinySTM) or cache-line level locking (e.g., McRT-STM C/C++). Our sensitivity analysis shows that a lock granularity of four words outperforms both word-level and cache line-level locking by 4% and 5% respectively across all benchmarks we considered.

To summarize, the main contributions of this paper are (1) the design and implementation of an STM that performs particularly well with large-scale complex transactional workloads while having good performance in small-scale ones, and (2) an extensive experimental evaluation of STM strategies and implementations from the perspective of complex applications with mixed workloads.

The rest of the paper is organized as follows. In Section 2, we give a short overview of STM design space and benchmarks. We then present SwissTM in Section 3. In Sections 4 and 5, we present the results of our experimental evaluation: first, we compare the performance of SwissTM to that of TL2, TinySTM, and RSTM, and, second, we evaluate the individual impact of the design choices underlying SwissTM.

2. Background

Transactional memory was first proposed in hardware (HTM) [23]. So far, most HTMs support only limited-size transactions and often do not ensure transaction progress upon specific system events, e.g., interrupts, context switches, or function calls [9]. While there have been proposals for truly dynamic HTMs (e.g. [3, 32]), it is very likely that actual HTM implementations will still have some of these limitations. Hybrid approaches either execute short transactions in hardware and fall back to software for longer ones (e.g., [26]), or accelerate certain operations of an STM in hardware. This work focuses on pure software solutions (STM) [37]. In this section, we survey some distinctive features of STMs and discuss the three representative STMs we focus on in our evaluation: RSTM [27], TL2 [10], and TinySTM [31] (see [24] for a full survey). We also give a short description of the benchmarks used in our experiments.

2.1 STM Design Space

The main task of an STM is to *detect* conflicts among concurrent transactions and *resolve* them. Deciding what to do when conflicts arise is performed by a (conceptually) separate component called a *contention manager* [22]. A concept closely related to conflict detection is that of *validation*. Validating a transaction consists of checking its read set (i.e., the set of locations[2] the transaction has already read) for consistency.

Two classes of STMs can be distinguished, *word*-based and *object*-based, depending on the granularity at which they perform logging. RSTM is object-based while TL2 and TinySTM are word-based. There are also two general classes of STM implementations: *lock*-based and *obstruction-free*. Lock-based STMs, first proposed in [19, 12], implement some variant of the two-phase locking protocol [13]. Obstruction-free STMs [22] do not use any blocking mechanisms (such as locks), and guarantee progress even when some of the transactions are delayed. RSTM (version 3) is obstruction-free, while TL2 and TinySTM internally use locks.

Conflict detection. Most STMs employ the single-writer-multiple-readers strategy; accesses to the same location by concurrent transactions *conflict* when at least one of the accesses is a write (update). In order to commit, a transaction T must eventually *acquire* every location x that is updated by T. Acquisition can be *eager*, i.e., at the time of the first update operation of T on x, or *lazy*, i.e., at the commit time of T. A transaction T that reads x can be either *visible* or *invisible* [27] to other transactions accessing x. When T is invisible, T has the sole responsibility of detecting conflicts on x with transactions that write x concurrently, i.e., validating its read set. The time complexity of a basic validation algorithm is proportional to the size of the read set, but can be boosted with a global commit counter heuristic (RSTM), or a time-based scheme [10, 31] (TL2 and TinySTM).

A *mixed invalidation* conflict detection scheme (first proposed in [39]) eagerly detects write/write conflicts while lazily detecting read/write conflicts (it is a mix between pure lazy and pure eager schemes). A similar conflict detection scheme is provided by more general (but also more expensive) *multi-versioning* schemes used in LSA-STM [33] and JVSTM [7]. Mixed invalidation, which underlies SwissTM, has never been used with lock-based or word-based STMs, nor has it been evaluated with any large-scale workload.

RSTM supports lazy and eager acquisition, as well as visible and invisible reads (i.e., four algorithm variants). TL2 and TinySTM use, respectively, lazy and eager acquisition. Both TL2 and TinySTM employ invisible reads.

Contention management. The contention manager decides what a given transaction (*attacker*) should do in case of a conflict with another transaction (*victim*). Possible outcomes are: aborting the attacker, aborting the victim, or forcing the attacker to retry after some period.

The simplest scheme (which we call *timid*) is to always abort the attacker (possibly with a short back-off). This is the default scheme in TL2 and TinySTM. More involved contention managers were proposed in [41, 36, 16], and are provided with RSTM. They can also be combined at run-time [15]. *Polka* [41] assigns every transaction a priority that is equal to the number of objects the transaction accessed so far. Whenever the attacker waits, its priority is temporarily increased by one. If the attacker has a lower priority than the victim, it will be forced to wait (using exponential back-off to calculate the wait interval), otherwise the victim gets aborted. *Greedy* assigns each transaction a unique, monotonically increasing timestamp on its start. The transaction with the lower timestamp always wins. An important property of Greedy is that, unlike other contention managers we mention, it avoids starvation of transactions. Polka has been shown to provide best performance in smaller-scale benchmarks previously [41], while our experiments show that Greedy performs better in large-scale workloads (Section 5). *Serializer* is very similar to Greedy except that it assigns a new timestamp to a transaction on every restart, and thus does not prevent starvation or even livelocks of transactions.

2.2 STM Benchmarks

In this section, we give an overview of the benchmarks we use in our experiments. These represent a large spectrum of workload types: from simple data structures with small transactions (the red-black tree microbenchmark) to complex applications with possibly long transactions (STMBench7). All the benchmarks we used are implemented in C/C++.

STMBench7. STMBench7 [18] is a synthetic benchmark which workloads aim at representing realistic, complex, object-oriented applications that are an important target for STMs. STMBench7 exhibits a large variety of operations (from very short, read-only operations to very long ones that modify large parts of the data structure) and workloads (from workloads consisting mostly of read-only transactions to write-dominated ones). The data structure used by STMBench7 is many orders of magnitude larger than in other typical STM benchmarks. Also, its transactions are longer and access larger numbers of objects.

STMBench7 is inherently object-based and its implementations also use standard language libraries. A thin wrapper, described in [11], is thus necessary to use STMBench7 with word-based STMs (TL2, TinySTM, and SwissTM).

STAMP. STAMP [8] is a TM benchmarking suite that consists of eight different transactional programs and ten workloads.[3] STAMP applications are representative of various real-world workloads, including bioinformatics, engineering, computer graphics, and machine learning. While STAMP covers a broad range of possible STM uses, its does not involve very long transactions, such as those that might be produced by average, non-expert programmers or

[2] These are memory words in word-based STMs and objects in object-based STMs.

[3] We used STAMP version 0.9.9.

generated automatically by a compiler along the lines of [1]. Furthermore, some STAMP algorithms (e.g., *bayes*) split logical operations into multiple transactions and use intricate programming techniques that might not be representative of average programmers' skills.

Lee-TM. Lee-TM [4] is a benchmark that offers large, realistic workloads and is based on Lee's circuit routing algorithm. The algorithm takes pairs of points (e.g., of an integrated circuit) as its input and produces non-intersecting routes between them. While transactions of Lee-TM are significant in size, they exhibit very regular access patterns—every transaction first reads a large number of locations (searching for suitable paths) and then updates a small number of them (setting up a path). Moreover, the benchmark uses very simple objects (each can be represented as a single integer variable). It is worth noting that STAMP contains an application (called *labyrinth*) that uses the same algorithm as Lee-TM. However, Lee-TM uses real-world input sets that make it more realistic than *labyrinth*. Lee-TM distribution includes two input data sets: *memory* and *main* circuit boards.

Red-black tree. The prevailing way of measuring the performance of STMs has been through microbenchmarks. The widely used (first in [22]) red-black tree microbenchmark consists of short transactions that insert, lookup, and remove elements from a red-black tree data structure. Short and simple transactions of microbenchmarks are good for testing mechanics of STM itself and comparing low-level details of various implementations.

3. SwissTM

SwissTM is a lock-based STM that uses invisible reads and counter based heuristics (the same as in TinySTM and TL2). It features eager write/write and lazy read/write conflict detection, as well as a two-phase contention manager with random linear back-off. The API of SwissTM is word-based, as it enables transactional access to arbitrary memory words. SwissTM uses a redo-logging scheme (partially to support its conflict detection scheme).

3.1 Programming model

Similarly to most other STM libraries, SwissTM guarantees opacity [17]. Opacity is similar to serializability in database systems [30]. The main difference is that all transactions always observe consistent states of the system. This means that transactions cannot, e.g., use stale values, and that they do not require periodic validation or sandboxing to prevent infinite loops or crashes due to accesses to inconsistent memory states.

SwissTM is a weakly atomic STM, i.e., it does not provide any guarantees for the code that accesses the same data from both inside and outside of transactions. SwissTM is not privatization safe [38]. This could make programming with SwissTM slightly more difficult in certain cases, but did not affect us, as none of the benchmarks we use requires privatization-safe STM.

When programming with SwissTM, programmers have to replace all memory references to shared data from inside transactions with SwissTM calls for reading and writing memory words. The programming model can be improved by using an STM compiler (as in e.g. [21, 2, 14, 29]). While the compiler instrumentation can degrade performance due to over-instrumentation [42] and possibly even change the characteristics of the workload slightly (e.g. numbers and ratio of transactional read and write operations), the compiler instrumentation remains a largely orthogonal issue to the performance of an STM library.

Other three STMs we compare to in our experiments provide the same semantical guarantees as SwissTM. Also, strengthening the guarantees (as described in Section 6) would have a similar performance impact on all STMs we use.

3.2 Algorithm

We give the pseudo-code of SwissTM in Algorithm 1. The algorithm invokes contention manager functions (*cm-**), which are defined in Algorithm 2 and described below. All transactions share a global commit counter *commit-ts* incremented by every non-read-only transaction upon commit. Each memory word *m* is mapped to a pair of locks in a global *lock table*: *r-lock* (read) and *w-lock* (write). Lock *w-lock* is acquired by a writer *T* of *m* (eagerly) to prevent other transactions from *writing* to *m*. Lock *r-lock* is acquired by *T* at commit time to prevent other transactions from *reading* word *m* and, as a result, observing inconsistent states of words written by *T*. In addition, when *r-lock* is unlocked, it contains the version number of *m*. Every transaction *T* has a *transaction descriptor tx* that contains (among other data): (1) the value of *commit-ts* read at the start or subsequent validation of *T*, and (2) read and write logs of *T*.

Transaction start. Every transaction *T*, upon its start, reads the global counter *commit-ts* and stores its value in *tx.valid-ts* (line 2).

Reading. When reading location *addr*, transaction *T* first reads the value of *w-lock* to detect possible read-after-write cases. If *T* is the owner of *w-lock*, then *T* can return the value from its write log immediately, which is the last value *T* has written to *addr* (line 6). Otherwise, i.e., when some other transaction owns *w-lock* or when *w-lock* is unlocked, *T* reads the value of *r-lock*, then the value of *addr*, and then again the value of *r-lock*. Transaction *T* repeats these three reads until (1) two values of *r-lock* are the same, meaning that *T* has read consistent values of *r-lock* and *addr*, and (2) *r-lock* is unlocked (lines 8–15). When *r-lock* is unlocked, it contains the current version *v* of *addr*. If *v* is lower or equal to the validation timestamp *tx.valid-ts* of *T* (which means that *addr* has not changed since *T*'s last validation or start), *T* returns the value at *addr* read in line 18. Otherwise, *T* revalidates its read set. If the revalidation does not succeed, *T* rolls back (line 17). If it succeeds, the read operation returns and *T* extends its validation timestamp *tx.valid-ts* to the current value of *commit-ts* (line 56).

Writing. Whenever some transaction *T* writes to a memory location *addr*, *T* first checks if *T* is the owner of the lock *w-lock* corresponding to *addr*. If it is, *T* updates the value of *addr* in its write log and returns (lines 21–23). Otherwise, *T* tries to acquire *w-lock* by atomically replacing, using a compare-and-swap (CAS) operation, value `unlocked` with the pointer to the *T*'s write log entry that contains the new value of *addr* (line 29). If CAS does not succeed, *T* asks the contention manager whether to rollback and retry or wait for the current owner of the lock to finish (line 26). In order to guarantee opacity, *T* has to revalidate its read set if the current version of *addr* (contained in *r-lock*) is higher than its validity timestamp *tx.valid-ts* (lines 31–32).

Validation. To validate itself, *T* compares the versions of all memory locations read so far to their versions at the point they were initially read by *T* (lines 51–52). These versions are stored in *T*'s read log. If there is a mismatch between any version numbers, the validation fails (line 52).

Commit. A read-only transaction *T* can commit immediately, as its read log is guaranteed to be consistent (line 35). A transaction *T* that is not read-only first locks all read locks of memory locations *T* has written to (line 36). Then, *T* increments *commit-ts* (line 37) and re-validates its read log. If the validation does not succeed, *T* rollbacks and restarts (lines 38–41). Upon successful validation, *T* traverses its write set, updates values of all written memory locations, and releases the corresponding read and write locks (lines 42–45). When releasing read locks, *T* writes the new value of *commit-ts* to those locks.

Algorithm 1: Pseudo-code representation of SwissTM.

```
1  function start(tx)
2      tx.valid-ts ← commit-ts;
3      cm-start(tx);

4  function read-word(tx, addr)
5      (r-lock, w-lock) ← map-addr-to-locks(addr);
6      if is-locked-by(w-lock, tx) then return get-value(w-lock, addr);
7      version ← read(r-lock);
8      while true do
9          if version = locked then
10             version ← read(r-lock);
11             continue;
12         value ← read(addr);
13         version2 ← read(r-lock);
14         if version = version2 then break;
15         version2 ← version;
16     add-to-read-log(tx, r-lock, version);
17     if $version > tx.valid\text{-}ts$ and not extend(tx) then rollback(tx);
18     return value;

19 function write-word(tx, addr, value)
20     (r-lock, w-lock) ← map-addr-to-locks(addr);
21     if is-locked-by(w-lock, tx) then
22         update-log-entry(w-lock, addr, value);
23         return;
24     while true do
25         if is-locked(w-lock) then
26             if cm-should-abort(tx, w-lock) then rollback(tx);
27             else continue;
28         log-entry ← add-to-write-log(tx, w-lock, addr, value);
29         if compare&swap(w-lock, unlocked, log-entry) then
30             break;
31     if $read(r\text{-}lock) > tx.valid\text{-}ts$ and not extend(tx) then
32         rollback(tx);
33     cm-on-write(tx);

34 function commit(tx)
35     if is-read-only(tx) then return;
36     for log-entry in tx.read-log do write(log-entry.r-lock, locked);
37     ts ← increment&get(commit-ts);
38     if $ts > tx.valid\text{-}ts + 1$ and not validate(tx) then
39         for log-entry in tx.read-log do
40             write(log-entry.r-lock, log-entry.version);
41         rollback(tx);
42     for log-entry in tx.write-log do
43         write(log-entry.addr, log-entry.value);
44         write(log-entry.r-lock, ts);
45         write(log-entry.w-lock, unlocked);

46 function rollback(tx)
47     for log-entry in tx.write-log do
48         write(log-entry.w-lock, unlocked);
49     cm-on-rollback(tx);

50 function validate(tx)
51     for log-entry in tx.read-log do
52         if $log\text{-}entry.version \neq read(log\text{-}entry.r\text{-}lock)$ and not
           is-locked-by(log-entry.r-lock, tx) then return false;
53     return true;

54 function extend(tx)
55     ts ← read(commit-ts);
56     if validate(tx) then tx.valid-ts ← ts; return true;
57     return false;
```

Errata: It should be tx.write-log in both places.

Algorithm 2: Pseudo-code of the two-phase contention manager (W_n is a constant)

```
1  function cm-start(tx)
2      if not-restart(tx) then tx.cm-ts ← ∞ ;

3  function cm-on-write(tx)
4      if $tx.cm\text{-}ts = \infty$ and $size(tx.write\text{-}log) = W_n$ then
           tx.cm-ts ← increment&get(greedy-ts) ;

5  function cm-should-abort(tx, w-lock)
6      if $tx.cm\text{-}ts = \infty$ then return true;
7      lock-owner = owner(w-lock);
8      if $lock\text{-}owner.cm\text{-}ts < tx.cm\text{-}ts$ then return true;
9      else abort(lock-owner); return false;

10 function cm-on-rollback(tx)
11     wait-random(tx.succ-abort-count);
```

Rollback. On rollback, transaction T releases all write locks it holds (lines 47–48), and then restarts itself.

Contention management. We give the pseudo-code of our two-phase contention manager in Algorithm 2. The contention manager gets invoked by Algorithm 1 (1) at transaction start (*cm-start* in line 3), (2) on a write/write conflict (*cm-should-abort* in line 26), (3) after a successful write (*cm-on-write* in line 33), and (4) after restart (*cm-on-rollback* in line 49). Every transaction, upon executing its ${W_n}^{\text{th}}$ write (where we set W_n to 10), increments global counter *greedy-ts* and stores its value in *tx.cm-ts* (line 4). Hence, short transactions (those that execute less than W_n writes) do not access *greedy-ts* that would otherwise become a memory hot spot—this reduces contention and the number of cache misses. Transactions that have already incremented *greedy-ts* are in the second phase of the contention management scheme, and others are in the first phase. Upon a conflict, a transaction that is still in the first phase gets restarted immediately (line 6). If both conflicting transactions are already in the second phase, the transaction with the higher value of *cm-ts* is restarted (lines 8–9). This prioritizes transactions that have performed more work. Conceptually, transactions in the first phase have an infinite value of *cm-ts* (set in line 2). This means that (longer) transactions, which are in the second phase, have higher priority than (short) transactions that are in the first phase. After restarting, transactions are delayed using a randomized back-off scheme (line 11). This reduces probability of having some transaction aborted many times repeatedly because of the same conflict.

3.3 Implementation Highlights

We implemented SwissTM in C++ (g++ 4.0.1 compiler). We used the (fairly portable) atomic_ops library [5] for atomic operations implementation. Currently, SwissTM works on 32-bit x86 Linux 2.6.x and OS X 10.5 platforms (64-bit port is in progress).

Lock table. To map memory word m to a lock table entry, we take the address a of m, shift it to the right by 4 (it would be 5 with 64-bit words). This makes each lock map to consecutive four memory words (we empirically selected this value, as explained in Section 5). Then, we set all high order bits to zero. As the lock table contains 2^{22} entries in our implementation, we just perform logical AND operation between shifted address and $2^{22} - 1$ to get the index into the table. Figure 1 depicts the mapping scheme. Having multiple consecutive memory words mapped to the same lock table entry can result in false conflict, when unrelated memory words get locked together, but this does not cause any problems in practice.

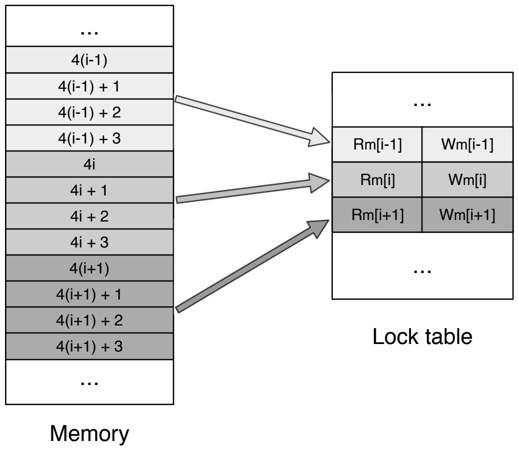


Figure 1. Mapping of memory words to lock table entries.

Every lock is implemented as a single memory word *w*. The write lock is equal to 0 in its unlocked state and contains a pointer to the corresponding write log entry when locked. Acquiring a write lock is done using a compare-and-swap (CAS) operation. When releasing write lock *w*, transactions simply write 0 to *w*. The read lock has its least significant bit set to 0 when unlocked, while other bits store the version number of memory locations corresponding to *w*. When locked, the read lock is equal to 1. Both locking and unlocking of read locks is performed by simply writing a new value to *w* (CAS is not used), as only the transaction that already acquired a write lock can acquire the corresponding read lock.

4. Evaluating SwissTM

We compare the performance of SwissTM to that of TL2, TinySTM, and RSTM.

We performed all measurements on a 4-processor dual-core AMD Opteron 8216 2.4 GHz 1024 KB cache machine with 8 GB of RAM, running Linux operating system. This provided us with 8 cores to experiment on. All results were averaged over multiple runs, where the length and the number of runs were chosen to reduce variations in collected data. We typically used 20 runs for STMBench7 and STAMP, 10 runs for LeeTM and 80 runs for the red-black tree microbenchmark. We used the TL2 x86 implementation provided with the STAMP benchmark suite (version 0.9.5). We were not able to use the TL2 implementation from original authors as it does not support the x86 architecture. Also, the original TL2 does not support transactional memory management in a straightforward manner, and is, because of that, difficult to use with benchmarks we had at our disposal, without significant changes to the benchmark code. While we did not use the original TL2 implementation in a setting that it was primarily designed for, we believe that the TL2 x86 port we used is the best representative of the TL2 algorithm and design available for the x86 architecture. The experiments were performed with the RSTM (version 3) and TinySTM (version 0.9.5) implementations available from respective sites. Unless stated otherwise, we configured RSTM to use eager conflict detection, invisible reads with the commit counter heuristic, and the Polka contention manager. We used default configurations of TL2 (i.e., lazy conflict detection, GV4) and TinySTM (i.e., encounter-time locking, timid contention manager).

STMBench7. Figure 2 shows the performance of SwissTM, TL2, TinySTM, and RSTM with STMBench7. We configured RSTM to use the Serializer contention manager, as this gave the best performing RSTM configuration in STMBench7. SwissTM significantly outperforms all other STMs for both read-dominated and read-write workloads, while also achieving superior scalability. SwissTM also outperforms other STMs in high-contention write-dominated workload, but it is only marginally faster than TinySTM.

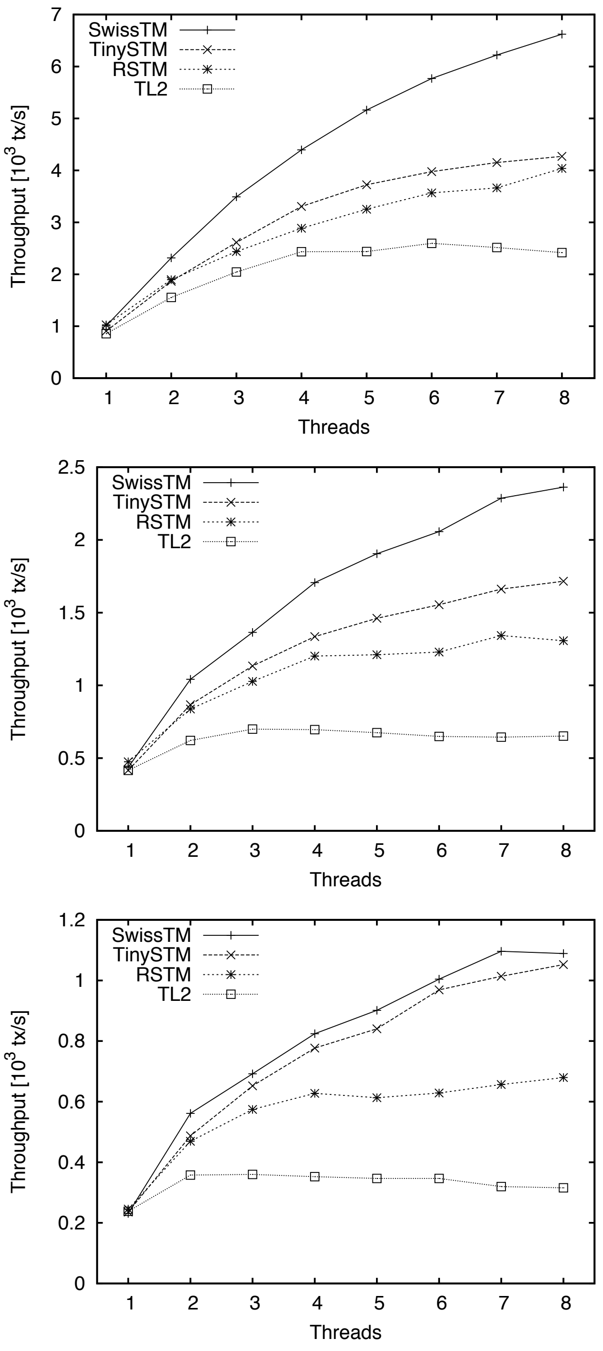
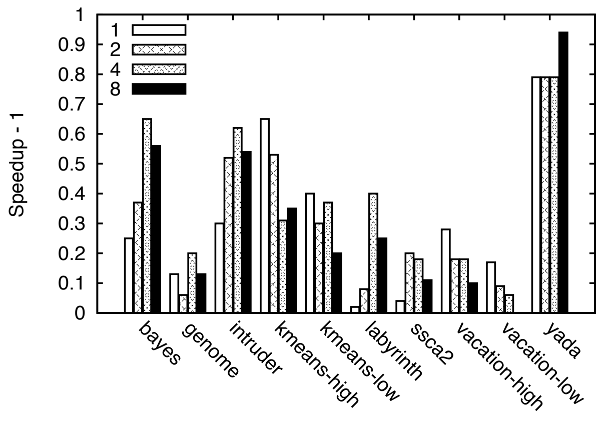


Figure 2. Throughput of SwissTM, RSTM, TL2, and TinySTM with STMBench7; top to bottom: read-dominated, read-write, and write-dominated workload

The main reason for the good performance of SwissTM is (a) its optimism in detecting read/write conflicts when compared to RSTM and TinySTM, and (b) its conservatism in detecting write/write conflicts when compared to TL2. The contention man-



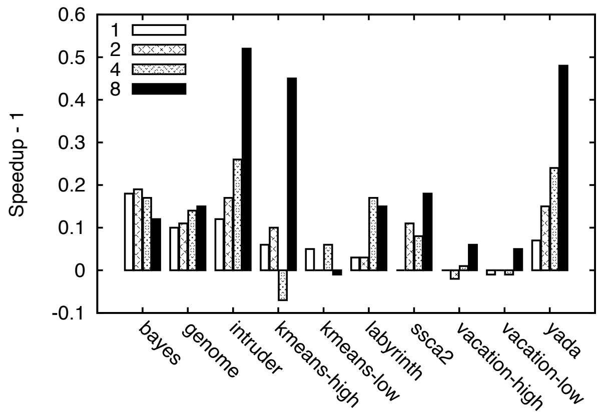
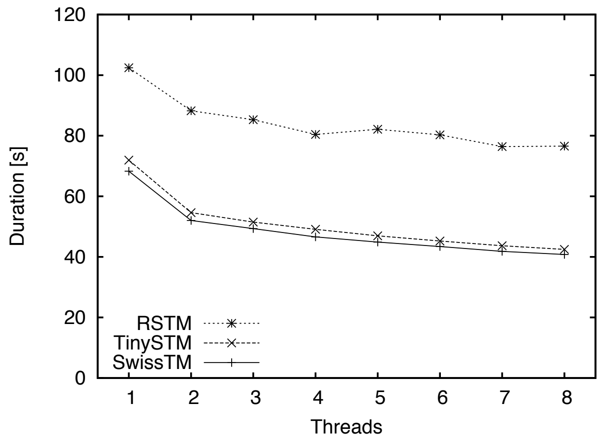


Figure 3. SwissTM vs. TL2 (top) and TinySTM (bottom) in STAMP. The figure conveys the speedup of SwissTM compared to TL2 and TinySTM (with 1 subtracted, i.e., positive numbers mean that SwissTM is faster and negative that it is slower) for 1, 2, 4, and 8 threads.



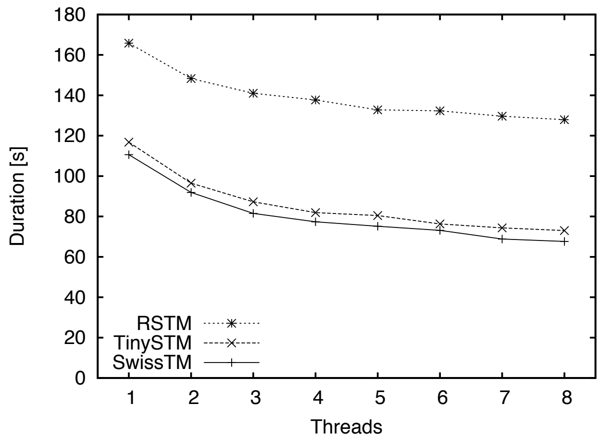


Figure 4. Execution time of SwissTM, TinySTM and RSTM in the Lee-TM benchmark; top is *memory*, bottom is *main* circuit board input data set

agement scheme used in SwissTM also helps boost performance, as we illustrate in Section 5.

TL2 performs poorly even in the read-dominated workload—it does not scale after 4 threads. Its performance gets even worse with higher contention. The main reason for this is the lazy conflict detection scheme of TL2, which wastes more work of transactions than the eager write/write conflict detection of other STMs.

STAMP. Figure 3 compares the performance of SwissTM, TL2, and TinySTM in the STAMP benchmark suite workloads.[4] SwissTM outperforms TL2 in all STAMP workloads, for all thread counts, excluding the *vacation* benchmark under low contention where TL2 and SwissTM have the same performance. SwissTM outperforms TL2 by over 50% with eight threads for the *bayes*, *intruder*, and *yada* benchmarks (being almost twice as fast as TL2 in *yada*), and by over 20% in *kmeans* (both variants) and *labyrinth*, while being about 10% faster than TL2 in *genome*, *ssca2*, and *vacation* under high contention. SwissTM outperforms TinySTM in ten STAMP workloads with eight threads, except for the *kmeans* benchmark under low contention where TinySTM has a slightly better performance (1% of difference). SwissTM outperforms TinySTM by over 45% with eight threads in *intruder*, *kmeans* under high contention, and *yada*, and by over 12% in *bayes*, *genome*, *labyrinth*, and *ssca2*, while being about 5% faster in *vacation* (both variants). SwissTM has good performance with lower thread counts, while scaling well as the number of concurrent threads increases. To summarize, SwissTM outperforms both TL2 and TinySTM in all STAMP benchmark workloads.

Lee-TM. Figure 4 compares the performance of SwissTM, RSTM, and TinySTM in the Lee-TM benchmark.[5] RSTM has the lowest performance mainly because of higher single object access overheads (objects in Lee-TM are very simple—consisting of a single integer variable). SwissTM and TinySTM have very similar performance, although SwissTM is faster by a small margin for all thread counts.

Red-black tree. Finally, Figure 5 compares the performance of SwissTM, TL2, TinySTM, and RSTM in the commonly used red-black tree microbenchmark. RSTM delivers significantly lower performance than other three STMs due to its high overheads on single memory location accesses. These low-level overheads have most significant impact in microbenchmarks like this one. This is precisely the reason while SwissTM, that uses two locks for each

[4] There is no RSTM implementation of STAMP, due to API incompatibility. RSTM provides an object-based API, while STAMP uses word-based API. This makes it difficult to simply plug-in RSTM into current STAMP version.

[5] We were not able to get Lee-TM to run with TL2 (we suspect there might be a bug in the x86 TL2 port).

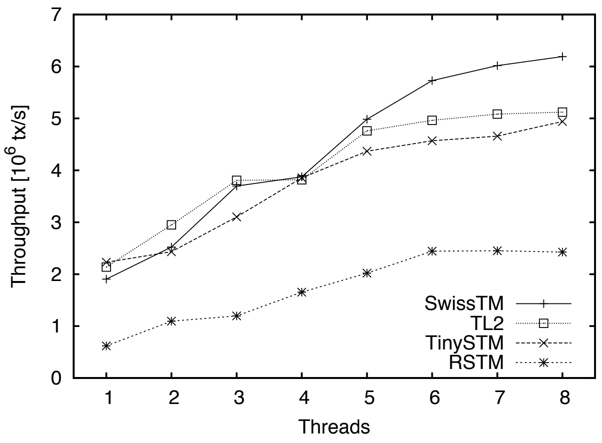


Figure 5. Throughput of SwissTM, TL2, TinySTM, and RSTM on red-black tree with range of 16384 and 20% of update operations

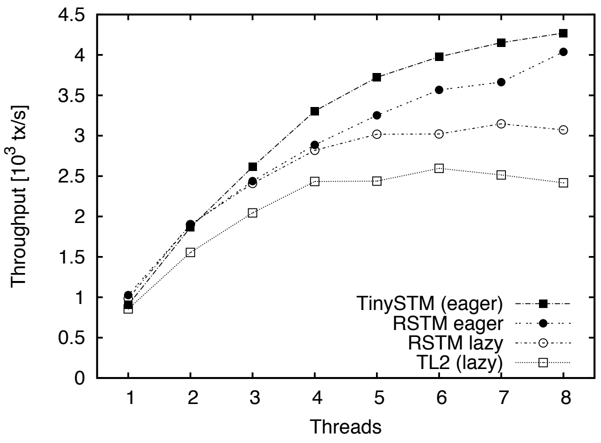


Figure 7. Throughput of eager and lazy conflict detection schemes in read-dominated STMBench7

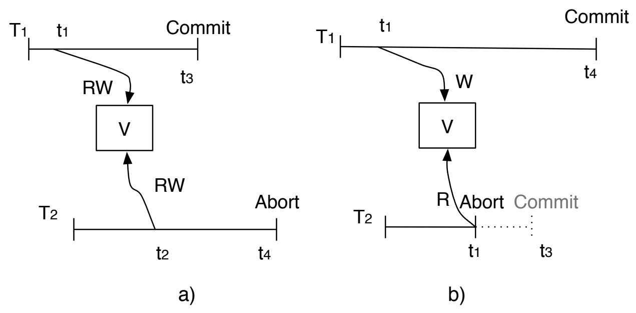


Figure 6. Disadvantages of lazy (left) and eager (right) conflict detection strategies

memory location, has lower performance than TL2 and TinySTM, which use a single lock, in single thread executions. It is worth noting that a red-black tree is the only benchmark for which slightly higher overheads of using two locks have more than negligible performance impact. SwissTM outperforms both TL2 and TinySTM when there are more than four threads and exhibits better scalability.

5. Dissecting the SwissTM Experience

In this section, we evaluate individually the design choices underlying SwissTM: its conflict detection strategy, the two-phase contention manager, and the granularity of the lock table.

Conflict detection. Current state-of-the-art STMs typically detect both read/write and write/write conflicts in the same way—either as soon as conflicts occur (eagerly, e.g., TinySTM, McRT-STM and Bartok STM), or at commit time (lazily, e.g., TL2). Detecting conflicts eagerly helps avoid wasting work of transactions that are doomed to abort after a conflict. Lazy conflict detection, however, is more optimistic and gives transactions more possibilities to commit. For example, Figure 6a depicts an execution of an STM that uses lazy conflict detection. There, transaction T_2 spends time between t_3 (commit time of T_1) and t_4 (commit time of T_2) performing work that is doomed to be roll-backed. The period between t_3 and t_4 can be significant with long transactions. It is worth noting that both T_1 and T_2 could commit with a lazy conflict detection STM, if they both only write to V. However, pure write/write conflicts are typically rare, as transactions usually first read some data and then subsequently update it. Because of this, lazy conflict detection STMs react too slowly to write/write conflicts (which are good signs that transactions cannot proceed in parallel) and results in transactions performing work that has to be rolled back later. Figure 6b, gives an example execution of an STM that uses eager conflict detection. There, transaction T_2 has to wait until time t_4 before continuing, although it could commit already at time t_3 if lazy scheme was used. The waiting time of T_2 might be significant if T_1 is very long.

SwissTM takes the best of both strategies—it detects write/write conflicts eagerly and read/write conflicts lazily. This combined strategy is beneficial for complex workloads with long transactions because it (1) prevents transactions with write/write conflicts from running for a long time before detecting the conflict, and (2) allows short transactions having a read/write conflict with longer ones to proceed, thus increasing parallelism.

Figure 2 suggests that the mixed eager-lazy scheme gives better performance than pure eager scheme which, in turn, outperforms lazy scheme (Figure 7). This does not say what kind of workloads benefit most from the mixed scheme, and what part of the performance boost of SwissTM can be attributed to its two-phase contention manager (and not to the mixed conflict detection).

To answer this question, we modified slightly the Lee-TM benchmark. The performance of the original Lee-TM does not seem to be influenced by the choice of a conflict detection and contention management schemes, because the transactions in Lee-TM are highly regular—they first read and then write. We introduce a small irregularity in Lee-TM by adding a single object O_c that every transaction reads at its start. A small ratio R of transactions (chosen randomly) also updates O_c, causing a read/write conflict with all the other transactions. The contention manager used by SwissTM does not provide a lot of benefit in this case, as the number of write/write conflicts introduced by this irregularity is not large (we keep R low in experiments).

Figure 8 compares the performance of TinySTM and SwissTM for R of 0%, 5%, and 20%. Due to its conflict detection scheme, the SwissTM performance degrades only slightly even when R is relatively high (20%). Also, SwissTM continues to scale well as the number of threads increases. On the other hand, the performance of TinySTM degrades significantly even when R is only 5%, while with R of 20% it stops scaling already at three threads.

We conclude here that applications exhibiting regular access patterns benefit the most from lowering single-location access costs

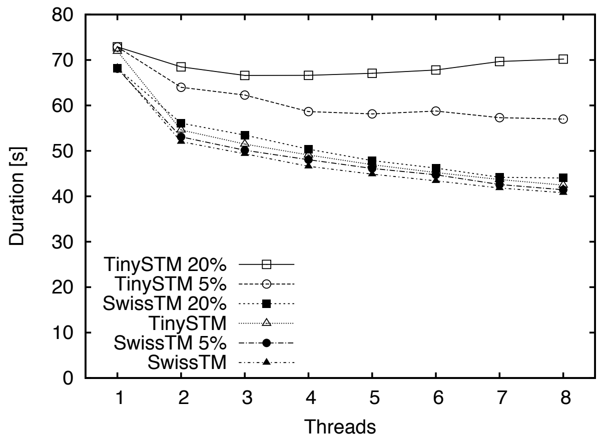


Figure 8. Execution time of SwissTM vs. TinySTM in "irregular" Lee-TM benchmark with *memory* circuit board input data set

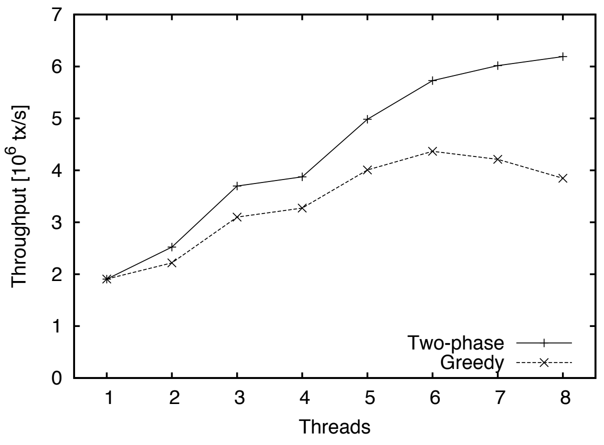


Figure 10. Throughput of the two-phase contention manager vs. Greedy (in SwissTM) with the red-black tree

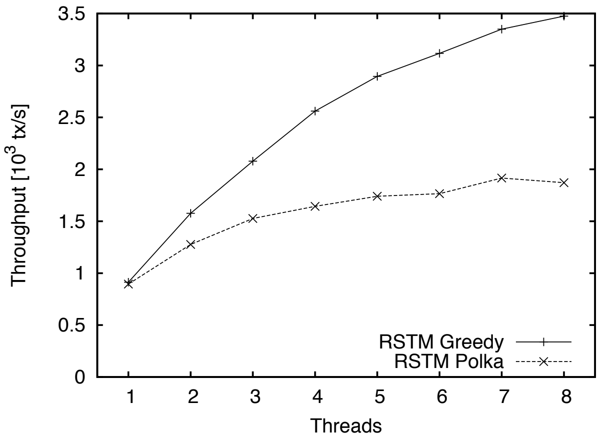


Figure 9. Throughput of Polka vs. Greedy (in RSTM) in the read-dominated workload of STMBench7

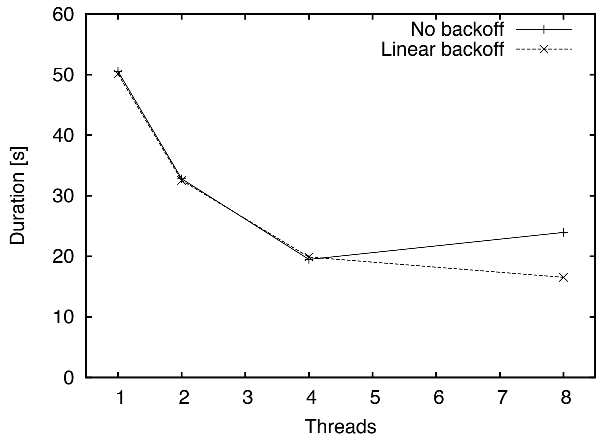


Figure 11. Execution time of backoff vs. no backoff (in SwissTM) in the STAMP *intruder* application

and are not significantly influenced by the conflict detection scheme itself. However, for applications where the access patterns introduce even small irregularities, especially those creating longer-lasting read/write conflicts, SwissTM's optimistic approach yields significant benefits.

Contention management. The contention manager gets invoked in SwissTM only on write/write conflicts. The reasoning here is simple—read/write conflicts are detected only at commit time of the writer, and it makes little sense for the reader to abort the writer that is already committing. This is why the reader waits until the writer (quickly) commits, before attempting to revalidate its read set.

Figure 9 shows that Greedy performs better than Polka (which was shown previously to perform very well in a range of smaller scale benchmarks [41]) in STMBench7, our larger-scale benchmark. However, Greedy performs poorly with short transactions, because all transactions increment a single shared counter at their start, which causes a lot of cache misses and significantly degrades performance and scalability (Figure 10). This problem is not noticeable with longer transactions as the overhead caused by cache misses is relatively small compared to the work of the transactions themselves. As shown in Figure 10, our two-phase contention manager overcomes this issue completely, improving both performance and scalability over Greedy. This is because it allows all short and read-only transactions to commit without incrementing the shared counter used by the Greedy algorithm, yet it provides the strong progress guarantees of Greedy for long ones.

It might seem beneficial to make transactions restart as soon as possible after conflicts that force them to rollback, as waiting just decreases the reaction time before the transaction re-executes. However, restarting immediately tends to increase contention on cache lines containing data that gets updated very frequently. Consequently, short back-offs after transaction rollbacks can improve performance. Figure 11 compares the performance of SwissTM in the STAMP *intruder* benchmark with and without the back-off scheme. (The *intruder* benchmark is a good example here, because it indeed contains a "hot spot": a high number of transactions dequeue elements from a single queue.) The figure shows that immediately restarting transactions after rollback causes scalability problem with eight threads. A simple randomized linear (in the number of successive aborts) back-off scheme resolves the scalability issue.

Finally, we evaluate the influence of our two-phase contention manager on the overall performance of SwissTM. Figure 12 shows that the two-phase contention manager improves performance by as

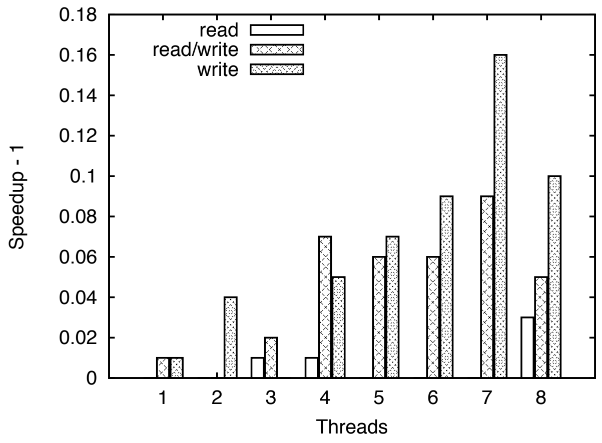


Figure 12. Speedup (with 1 subtracted) of the two-phase contention manager over the timid contention manager in SwissTM with STMBench7

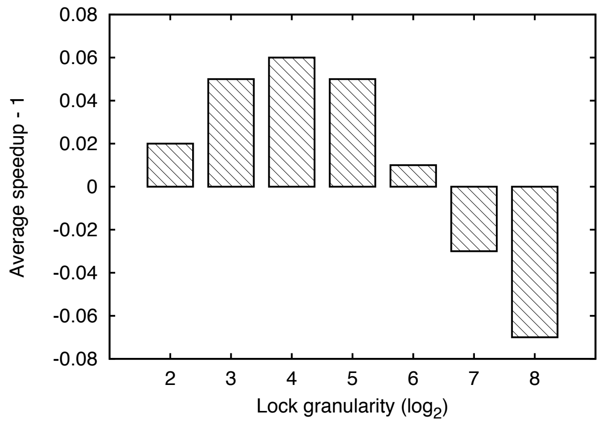


Figure 13. Average speedup across all benchmarks used (with one subtracted) of locking granularities from 2^2 to 2^8 compared to all other granularities, with 8 threads

much as 16% in high-contention workloads. Its influence is lower in the read-dominated workload, which is not surprising given that this workload is characterized by a small number of write/write conflicts.

Locking granularity. An important implementation choice in an STM is its lock table configuration, in particular the size of the memory stripe that gets mapped to the same (lock table) entry. Increasing the size of memory stripes reduces locking and validation time, due to the data access locality, but increases abort rates by introducing false conflicts when the memory stripe becomes too large. The optimal value for this parameter is application specific and we searched for the best value across all benchmarks we used.[6] Figure 13 depicts the average speedup (minus 1) of each logarithmic lock granularity compared to all the others at eight threads (32 bit word). The figure shows that the granularity of 2^4 bytes achieves the best performance, with 2^3 and 2^5 being slightly slower. It is interesting to note that the commonly used sizes of one word (2^2) and one cache line (2^6) have performance of 4% and 5% lower on average than the one we select. We give a breakdown of these differences across benchmarks in Table 2.

It is interesting to note here that, while using different lock granularities does impact performance, the impact of using coarser lock granularities is not significant enough to prevent SwissTM from scaling (e.g. due to increased number of false conflicts).

	Difference in performance		
Benchmark	2^4 vs. 2^2	2^4 vs. 2^6	2^2 vs. 2^6
bayes	0.16	0.81	0.57
genome	0.13	−0.03	−0.14
intruder	0	−0.04	−0.04
kmeans-high	0.19	0.4	0.18
kmeans-low	0.14	0.05	−0.08
labyrinth	−0.12	−0.09	0.04
ssca2	0	0	0
vacation-high	0.14	−0.03	−0.15
vacation-low	0.12	−0.05	−0.15
yada	0	0.12	0.12
red-black tree	−0.01	0	0.01
Lee-TM memory	0.01	−0.03	−0.04
Lee-TM main	0.02	−0.01	−0.02
STMBench7 read	0	−0.02	−0.02
STMBench7 read-write	−0.01	−0.03	−0.02
STMBench7 write	−0.04	−0.06	−0.02
Average	0.05	0.06	0.01

Table 2. Comparing three different lock granularities. Numbers represent relative speedups (with one subtracted) with 8 threads

6. Concluding Remarks

This paper presents SwissTM—an effective compilation of STM design choices for mixed workloads characterized by non-uniform, dynamic data structures and various transaction sizes. Those kinds of workloads are inherent to many applications that might be expected to significantly benefit from the STM paradigm and multicore architectures. SwissTM significantly outperforms state-of-the-art STMs in precisely such workloads, while also delivering good performance in smaller-scale scenarios.

Not surprisingly, the design of SwissTM is a result of trial-and-error. We reported in the paper on various choices that might have seemed natural, but revealed inappropriate. Besides those, we also experimented with nested transactions (closed nesting) and multi-versioning, but we could not see a clear advantage of those techniques in the considered workloads. Further experiments might be needed in this direction.

[6] Performance results presented in previous sections all use the same locking granularity of 2^4 bytes.

Improving the programming model. Two main directions along which we plan to improve the semantical guarantees of SwissTM are: (1) adding compiler support, and (2) making SwissTM privatization-safe. There exists a number of STM C/C++ compilers that have open interfaces supporting different STM libraries (e.g. [14, 29]) and we plan to integrate SwissTM with one of them. A conceptually simple algorithm for ensuring privatization-safety uses quiescence: every committing transaction T has to wait for all in-flight transactions to validate, commit, or abort. While this algorithm is simple, it would probably significantly impact performance of SwissTM [42] and we plan to investigate other options, possibly using techniques similar to [28] or [25].

7. Availability

SwissTM is open-source and available from: `http://lpd.epfl.ch/site/research/tmeval`. The source code of benchmarks used in the paper can be downloaded from the same location.

Acknowledgments

We would like to thank Hagit Attiya, Pascal Felber, Eshcar Hillel, and Nir Shavit, as well as the anonymous reviewers, for their helpful comments and discussions on the topic of this paper.

This work is funded by the Velox FP7 European project and the Swiss National Science Foundation grant 200021-116745/1.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.

[4] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Lee-TM: A non-trivial benchmark for transactional memory. In *ICA3PP*, 2008.

[5] The atomic_ops project. `http://www.hpl.hp.com/research/linux/atomic_ops`.

[6] C. Blundell, H. Cain, M. M. Michael, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, Sept. 2008.

[7] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT*, 2008.

[10] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[11] A. Dragojević, R. Guerraoui, and M. Kapałka. Dividing transactional memories by zero. In *TRANSACT*, 2008.

[12] R. Ennals. Efficient software transactional memory. Technical report, Intel Research Cambridge, Jan 2005.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[14] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süsskraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, 2007.

[15] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *DISC*, 2005.

[16] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC*, 2005.

[17] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *PPoPP*, 2008.

[18] R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys*, 2007.

[19] T. Harris and K. Fraser. Revocable locks for non-blocking programming. In *PPoPP*, 2005.

[20] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

[21] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI*, 2006.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[24] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan&Claypool, 2007.

[25] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT*, 2009.

[26] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT*, 2007.

[27] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT*, 2006.

[28] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP*, 2008.

[29] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA*, 2008.

[30] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.

[31] T. R. Pascal Felber and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[32] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA*, 2005.

[33] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

[34] RSTM home page. `http://www.cs.rochester.edu/research/synchronization/rstm`.

[35] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[36] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *CSJP*, 2004.

[37] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[38] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC)*, 2007.

[39] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC*, 2006.

[40] T. Sweeney. The next mainstream programming language: a game developer's perspective. Invited talk at POPL, 2006.

[41] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.

[42] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA*, 2008.