RobuSTM: A Robust Software Transactional Memory

Jons-Tobias Wamhoff[1], Torvald Riegel[1], Christof Fetzer[1], and Pascal Felber[2]

[1] Dresden University of Technology, Germany (*first.last@tu-dresden.de*)
[2] University of Neuchâtel, Switzerland (*first.last@unine.ch*)

Abstract. For software transactional memory (STM) to be usable in large applications such as databases, it needs to be *robust*, i.e., live, efficient, tolerant of crashed and non-terminating transactions, and practical. In this paper, we study the question of whether one can implement a robust software transactional memory in an asynchronous system. To that end, we introduce a system model – the *multicore system model* (MSM) – which captures the properties provided by mainstream multicore systems. We show how to implement a *robust software transactional memory* (RobuSTM) in MSM. Our experimental evaluation indicates that RobuSTM compares well against existing blocking and nonblocking software transactional memories in terms of performance while providing a much higher degree of robustness.

1 Introduction

Software transactional memory (STM) is a promising approach to help programmers parallelize their applications: it has the potential to simplify the programming of concurrent applications, when compared to using fine-grained locks. Our general goal is to investigate the use of STM in large software systems like application servers, databases, or operating systems. Such systems are developed and maintained by hundreds of programmers, and all that code lives in the same address space of the system's process. Ensuring the robustness of such applications requires the use of techniques that guarantee the recovery from situations in which individual threads crash or behave improperly (e.g., loop infinitely) while executing critical sections. For example, commercial databases guarantee such robustness using custom mechanisms for lock-based critical sections [12].

A system that uses transactions to perform certain tasks typically relies on their completion. Thus, a robust STM must guarantee that all *well-behaved* transactions will terminate within a finite number of steps. A transaction is *well-behaved* if it is neither *crashed* nor *non-terminating*. Both crashed and non-terminating transactions can interfere with the internal synchronization mechanism of the underlying STM implementation, possibly preventing other transactions from making progress if not handled correctly. A *crashed* transaction will stop executing prematurely, i.e., it executes a finite number of steps and stops before committing (e.g., due to failure of the associated thread). A *non-terminating* transaction executes an infinite number of steps without attempting to commit.

Note that a robust STM provides guarantees that are very similar to a *wait-free* STM, which guarantees to commit all concurrent transactions in a bound number of steps. Yet, the definition of the *wait-free* property requires the use of an asynchronous model of computation, but it has been shown recently [10] that one *cannot* implement a wait-free STM in an such a system model. However, current multicore systems provide stronger guarantees than those postulated in the asynchronous system model. Therefore, we try to answer the question whether one can implement a robust STM in today's multicore computer architectures.

In this paper, we introduce a new *multicore system model* (MSM). It is asynchronous in the sense that it does not guarantee any bounds on the absolute or relative speed of threads but additionally reflects the properties of mainstream multicore systems. We show that one can implement a robust STM (RobuSTM) in MSM that guarantees progress for individual threads. Our RobuSTM implementation exhibits performance comparable to state-of-the-art lock-based STMs on various types of benchmarks. Therefore, we not only show that one can implement *robust* STMs but also that one can implement them *efficiently*.

The paper is organized as follows: We first introduce MSM in Section 2. Section 3 presents the algorithm of RobuSTM. We evaluate our approach in Section 4 and discuss related work in Section 5. We conclude in Section 6.

2 System Model

Our multicore system model (MSM) satisfies the following nine properties. (1) A process consists of a non-empty set of threads that share an address space. (2) All non-crashed threads execute their program code with a non-zero speed. Neither the absolute nor the relative speed of threads is bounded. (3) Threads can fail by crashing. A crash can be caused by a programming bug or by a hardware issue. In the case of a hardware issue, we assume that the process crashes. In case of a software bug, only a subset of the threads of a process might crash. (4) We assume that STM is correctly implemented, i.e., crashes of threads are caused by application bugs and not by the STM itself. The motivation is that a STM has typically a much smaller code size that is reused amongst multiple applications. (5) A process can detect the crash of one of its threads. (6) Threads can synchronize using CAS and *atomic-or* operations (see below). (7) The state of a process is finite. (8) A thread can clone the address space of the process. (9) Each thread has a performance counter that counts the number of instructions it executed.

Software Transactional Memory. In our model, we assume that transactions are executed concurrently by threads. Within transactions, all accesses to shared state must be redirected to the STM and neither non-transactional accesses to global data nor external IO operations are permitted. If a thread failed to commit the transaction, it is retried (see Section 3). We further assume that transactions are non-deterministic and allow transactions to execute different code paths or access different memory locations during the retry.

Detection mechanisms. Modern operating systems permit the detection of thread crash failures. A thread can crash for various reasons like an uncaught exception. To detect the crash of a thread that is mapped to an operating system process, one can read the list of all processes that currently exist and check their status or search for missing threads [1]. The MSM assumes the existence of a thread crash detector that detects within a finite number of steps when a thread has crashed (i.e., the thread stopped executing steps) and will not wrongly suspect a correct thread to have crashed. For simplicity, in our implementation we assume that a signal handler is executed whenever a thread crashes.

Progress mechanisms. Like many concurrent algorithms, the MSM assumes the existence of a compare-and-swap (CAS) operation: `bool CAS(addr, expect, new)`. CAS atomically replaces the content of address `addr` with value `new` if the current value is `expect`. It returns `true` iff it stored `new` at `addr`. A CAS is often used in loops in which a thread retries until its CAS succeeds (see Figure 1). Note that sometimes such a loop might contain a contention manager to resolve a conflict with another thread but in the meantime a third thread might have successfully changed `addr`. In other words, a contention manager might not be able to ensure progress of an individual thread since this thread might have continuous contention with two or more other threads.

```
repeat
    expect = *addr;          ▷ Read current value
    new = function(expect);  ▷ Get new value
until CAS(addr, expect, new)
```

Fig. 1. While CAS is wait-free, there is no guarantee that the CAS will ever succeed, i.e., that the loop ever terminates.

```
repeat
    if has_priority() then ▷ Privileged priority
        atomic-or(addr, F);          ▷ Set fail bit
        expect = *addr;      ▷ Expect bit in CAS
    else                      ▷ All other threads
        expect = *addr & ∼F;         ▷ No fail bit
    end if
until CAS(addr, expect, new)
```

Fig. 2. Using an *atomic-or*, we can make sure the CAS of the privileged priority thread always succeeds.

The problem is that there is no guarantee that a thread will ever be successful in performing a CAS. To address this issue, the MSM assumes an *atomic-or* operation. Note that the x86 architecture supports such an operation: a programmer can just add a `LOCK` prefix to a logical `or` operation. It is guaranteed, that a processor will execute the *atomic-or* operation in a finite number of steps. Also note that such an operation does not exist on, for example, Sparc processors.

We use the *atomic-or* to ensure that each correct transaction will eventually commit. RobuSTM will select at most one thread with a privileged priority level in the sense that this thread should win all conflicts. To ensure that all CAS operations performed by a privileged thread succeed, it uses the *atomic-or* to make sure that all competing CASes fail. To do so, we reserve a bit (`F`) in each word that is used with a CAS (see Figure 2). If a privileged thread performs an *atomic-or* just before another thread tries to perform a CAS, the latter will fail because its expected value assumes the `F` bit to be cleared.

Our goal is not only to implement wait-free transactions in the face of crash failures, but also in the face of non-terminating transactions. We assume however that the STM code itself is well-behaved and only application code can crash or loop infinitely often. For tolerating non-terminating transactions, we assume two more mechanisms that can be found in current systems. First, the MSM assumes that we can *clone* a thread, i.e., the operating system copies the address space of a process (using copy-on-write) and the cloned thread executes in a new address space fully isolated from all threads of the original process. Second, the MSM assumes the existence of a performance counter that (1) counts the cycles executed by a thread, and (2) permits other threads to read this performance counter. The intuition of the performance counter is as follows. The privileged thread can keep its privilege for a certain number of cycles (measured by the performance counter), after which it is not permitted anymore to steal the locks of other threads. If we can prove that the thread is well-behaved and would have simply needed more time to terminate, we increase the time quantum given to the privileged thread. Since the state space of threads is finite (but potentially very large), there exists a finite threshold S such that each transaction will either try to commit in at most S steps, or it will never try to commit. The problem is how to determine an upper bound on this threshold for non-deterministic transactions (see Section 3). Our system ensures that non-terminating transactions are eventually isolated to ensure the other threads can make progress while ensuring that long running but correct transactions will eventually commit.

3 Design and implementation

Our STM algorithm runs in different modes. In this section, we first present the basic algorithm optimized for the good case with well-behaved transactions (Mode 1). When conflicts are detected and fairness is at stake, we switch to Mode 2 by prioritizing transactions. If the system detects a lack of progress, we switch to Mode 3 for dealing with crashed and non-terminating transactions. The mode is set for each transaction individually.

3.1 Why a Lock-based Design?

Our robust STM algorithm uses a lock-based design. The reason for basing our work on a blocking approach instead of an obstruction-free one is driven by performance considerations. Non-blocking implementations suffer from costly indirections necessary for meeting their obstruction-free progress guarantee [5, 4, 16]. Although many techniques known from blocking implementations were applied to avoid indirection under normal operation with little contention, indirection is still necessary when it comes to conflicts with transactions that are not well-behaved (see Section 5). Our own experiments (see Section 4) still show a superior performance of lock-based designs.

Several reasons can explain the good performance of blocking STMs. They have a simpler fast path and more streamlined implementations of the read/write operations with no extra indirection. In addition, the combination of invisible reads and time-based validation [17] provides significant performance benefits. In this paper, we use a C++ version of the publicly-available TinySTM [6] as the basis for our robust STM algorithm. TinySTM is an efficient lock-based implementation of the lazy snapshot algorithm (LSA) [17].

3.2 Optimizing for the Good Case

For completeness, we briefly recall here the basic algorithm used by TinySTM. Like several other word-based STM designs, TinySTM relies upon a shared array of *locks* to protect memory from concurrent accesses (see Figure 3). Each lock covers a portion of the address space. In our implementation, it uses a per-stripe mapping where addresses are mapped to locks based on a hash function.

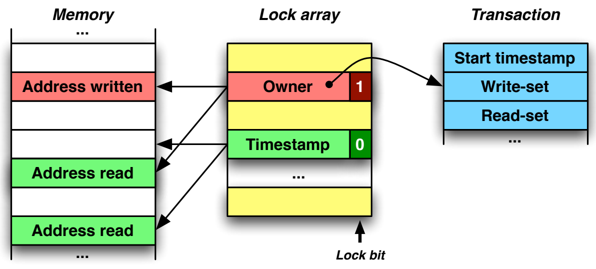

Fig. 3. Data structures for the lock-based design of TinySTM.

Each lock is the size of an address on the target architecture. Its least significant bit is used to indicate whether the lock has been acquired by some transaction. If it is free, the STM stores in the remaining bits a version number that corresponds to the commit timestamp of the transaction that last wrote to one of the memory locations covered by the lock. If the lock is taken, the STM stores in the remaining bits a pointer to an entry in the write-set of the owner transaction. Note that addresses point to structures that are word-aligned and their least significant bits are always zero on 64-bit architectures; hence one of these bits can safely be used as lock bit.

When writing to a memory location, a transaction first identifies the lock entry that covers the memory address and atomically reads its value. If the lock bit is set, the transaction checks if it owns the lock using the address stored in the remaining bits of the entry. In that case, it simply writes the new value into the transaction-private write set and returns. Otherwise, there is a conflict and the default contention management policy is to immediately abort the transaction (we will show how one can change this behavior to provide fairness shortly).

If the lock bit is not set, the transaction tries to acquire the lock using a CAS operation. Failure indicates that another transaction has acquired the lock in the meantime and the whole procedure is restarted. If the CAS succeeds, the transaction becomes the owner of the lock. This basic design thus implements visible writes with objects being acquired when they are first encountered.

When reading a memory location, a transaction must verify that the lock is neither owned nor updated concurrently. To that end, the transaction reads the lock, then the memory location, and finally the lock again (obviously, appropriate memory barriers are used to ensure correct ordering of accesses). If the lock is not owned and its value (i.e., version number) did not change between both reads, then the value read is consistent. If the lock is owned by the transaction itself, the transaction returns the value from its write set. Once a value has been read, LSA checks if it can be used to construct a consistent snapshot. If that is not the case and the snapshot cannot be extended, the transaction aborts.

Upon commit, an update transaction that has a valid snapshot acquires a unique commit timestamp from the shared clock, writes its changes to memory, and releases the locks (by storing its commit timestamp as version number and clearing the lock bit). Upon abort, it simply releases any lock it has previously acquired. Refer to [17] for more details about the LSA algorithm.

3.3 Progress and Fairness

An important observation is that the basic TinySTM algorithm does not provide liveness guarantees even when considering only well-behaved transactions. In particular, a set of transactions can repeatedly abort each other, thus creating livelocks. Furthermore, there is no fairness between transactions: a long-running transaction might be taken over and aborted many times by shorter update transactions, in particular if the former performs numerous invisible reads

To address these problems, we introduce two mechanisms that make up Mode 2. The first one consists of introducing "visible reads" after a transaction has aborted a given number of times because of failed validation (i.e., due to invisible reads). To that end, in addition to the `WR` bit used for writers, we use an additional `RD` bit in the lock metadata to indicate that a transaction is reading the associated data (see Figure 4). Using a different bit for visible readers allows more concurrency because an invisible reader is still allowed to read data that is locked in read mode. Other conflicts with visible readers are handled as for writers, i.e., only one transaction is allowed to proceed. The use of visible reads makes all conflicts detectable at the time data is accessed: a well-behaved transaction that wins all conflicts is guaranteed not to abort.

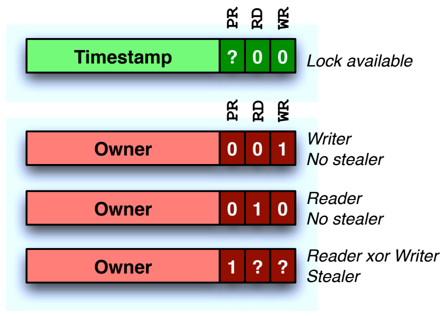

Fig. 4. Free and possibly reserved lock, owned lock, and owned lock with stealer.

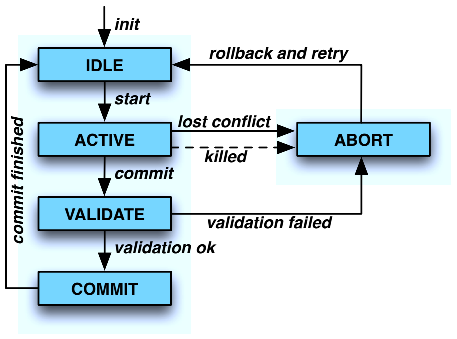

Fig. 5. States during the lifetime of a transaction.

This mechanism alone is not sufficient to guarantee neither progress nor fairness. Depending on the contention management strategy, transactions can repeatedly abort each other, or a transaction might always lose to others and never commit. To address the fairness problem, we need to be able to prioritize transactions and choose which one to abort upon conflict. That way, we can ensure that the transaction with the highest priority level wins all its conflicts.

A transaction that cannot commit in Mode 1, first switches to visible reads. If it still fails to commit after a given number of retries with visible reads enabled, it tries to enter a *privileged priority* level that accepts only one thread at a time. Entry into this priority level is guarded using Lamport's bakery algorithm [13] that provides fairness by granting permission in the order in which transactions try to acquire the bakery lock. Because the number of steps that transactions are allowed to execute with priority is limited (see Section 3.6), each acquire attempt will finish in a finite number of steps. The privileged thread can steal a lock from its current owner by *atomic-or*ing the `PR` bit to 1 before acquiring it. The bit indicates that a transaction is about to steal the lock (see Figure 4). As explained in Section 2, this will ensure that any other thread attempting to CAS the lock metadata will fail (because it expects the `PR` bit to be cleared), while the privileged thread will succeed.

3.4 Safe Lock Stealing

Due to the lock-based nature of our base STM, being able to safely steal locks from transactions is necessary to build a robust STM. Our system model eases this because it requires that STM code is well-behaved and only application code can crash or loop infinitely often.

To understand how lock stealing works, consider Figure 5 that shows the different states a transactions can take. The normal path of a transaction is through states `IDLE`, `ACTIVE` (when transaction has started), `VALIDATE` (upon validation when entering commit phase), and `COMMIT` (after successful validation when releasing locks). A transaction can abort itself upon conflict (from `ACTIVE` state), or when validation fails (from `VALIDATE` state).

An active transaction can also be forcefully aborted (or killed) by another transaction in privileged priority (dashed arrow in the figure). This happens when the privileged transaction tx wants to acquire a lock that is already owned, i.e., with the `RD` or `WR` bit set. In that case, tx first reserves this lock for the privileged transaction by *atomic-or*ing the `PR` bit to 1. This wait-free operation also ensures that other non-privileged transactions will notice the presence of tx and will not be able to acquire the lock or clear the `PR` bit anymore (see Figure 2). In RobuSTM, all lock acquire and release operations must be performed using `CAS`, which will fail for non-privileged transactions if the `PR` bit is set.

After reserving the lock, tx can continue with actually stealing the lock. It loads the value of the lock again and determines whether there was an owner transaction. If so and if the owner is in the `IDLE` state, it can just acquire the lock. If the owner is in the `VALIDATE` or `COMMIT` states, tx waits for the owner to either abort (e.g., because validation failed) or finish committing. We do not abort validating transactions because they might be close to successfully committing. Because we assume that STM code is well-behaved and because read sets are finite, commit attempts execute in a finite number of steps. Note that a successfully committed transaction releases only the locks whose `PR` bit is not set. This process works as long as there is at most one transaction in the privileged priority level that can steal locks.

If the owner transaction is in `ACTIVE` state, *tx* attempts to abort the owner by using `CAS` to change the state to `ABORT`. After that or if the owner is already in state `ABORT`, *tx* acquires the lock using `CAS` but while doing so expects the value that the lock had after the *atomic-or*. The `PR` bit is only used during lock stealing and is not set after *tx* acquired the lock. Transactions check whether they have been aborted within each STM operation (e.g., loads). Note that a transaction's state is versioned to avoid ABA issues on lock owners, i.e., *tx* can distinguish if the transaction that previously owned the lock aborted and retried while performing the lock stealing.

3.5 Dealing with Crashed Transactions

Using a traditional lock-based STM could lead to infinite delays in case a thread that has acquired some locks crashes. Because RobuSTM supports lock stealing, the crash of a transaction that is not in privileged priority level and that is not in the `COMMIT` state does not prevent other transactions from safe lock stealing introduced in Section 3.4.

RobuSTM makes use of the crash detector included in the MSM to deal with crashed transactions. In practice, events that cause a thread to crash (e.g., a segmentation fault or an illegal instruction) are detected by the operating system and a thread can request to be notified about such events by registering a signal handler. If a signal is received by a thread that indicates a crash, the thread will abort itself if it is in the `ACTIVE` state to speed up future acquisitions by other threads. If the thread is in the `COMMIT` state and already started writing its modifications to memory, it will finish comitting. The intention there is to always keep the shared state consistent and to reduce the contention on locks. Transactions in privileged priority level that encountered a thread crash additional release their priority.

3.6 Dealing with Non-Terminating Transactions

The main problem that we face when designing a robust STM is how to deal with non-terminating transactions as the locks they hold can prevent other transactions from making progress. Two different kinds of non-terminating transactions have to be distinguished: (1) transactions that are in `ACTIVE` state but stopped executing STM operations, and (2) transactions that still perform STM operations (e.g., in an infinite loop). Both correspond to non-crashed threads and never reach the `VALIDATE` state.

Let us first consider how RobuSTM handles threads that stopped executing STM operations (e.g., the thread is stuck in an endless loop). In the simplest case, the thread did not acquire any locks and thus does not prevent other threads from making progress and can be tolerated by the system. If the non-terminating transaction already acquired locks, it may run into a conflict with another thread. Eventually, the conflicting thread will reach the privileged priority level and again run into a conflict with the non-terminating transaction. It will then force the non-terminating transaction to abort and steals the lock.

Since the status of a thread is only checked during STM operations, the non-terminating transaction will not discover the update and will remain in the `ABORT` state. Other transactions that encounter a conflict with a transaction in `ABORT` state can simply steal the lock.

A non-terminating transaction that still performs STM operations will discover the update of its state to `ABORT`. It will roll back and retry its execution. If, during the retry, it becomes again a non-terminating transaction that owns locks, it will be killed and retried again. It can therefore enter the privileged priority level and still behave as a non-terminating transaction, hence preventing all other transactions from making progress because it wins all conflicts. Since we assume that the state of a computer is finite, for each well-behaved and privileged transaction tx there exists a maximum number of steps, $maxSteps$, such that tx will execute at most $maxSteps$ before trying to commit. $maxSteps$ is not known a priori and hence, we cannot reasonably bound the number of steps that a privileged transaction is permitted to execute without risking to prevent some well-behaved transactions from committing.

MSM permits us to deal with non-terminating transactions running at the privileged priority level as follows. The privileged thread t_h receives a budget of at most a finite number of steps but at least *quantum* steps, where *quantum* is a dynamically updated value. Initially, *quantum* is set to some arbitrary value that we assume to be smaller than $maxSteps$. The privileged thread t_h is forced to the `ABORT` state after *quantum* steps (determined with the help of the performance counters) and is removed from the privileged priority level.

If the formerly privileged transaction t_h notices that it has been aborted and exceeded its quantum, it clones its thread. The clone consists of a separate address space that is copied on write from the parent and a single thread that runs in isolation. Transactional meta data of all threads in the parent is copied with the address space. The clone then continues to execute the transaction in a *checker run* using the meta data to resolve conflicts. There are two cases to consider. (1) If t_h is well-behaved, it will terminate after running for, say, *childSteps*. At this point, the child will return *success* and the parent thread will increase *quantum* by setting it to a value of at least *childSteps*. Then, the parent thread will re-execute t_h at privileged priority with at least *quantum* steps. If the new *quantum* was not sufficient , e.g., because of non-determinism, it will be increased iteratively. (2) If t_h is not well-behaved, it will not terminate the checker run. In this case, the parent thread will wait forever for the child thread to terminate. Because the parent thread has aborted, it will not prevent any of the well-behaved threads from making progress.

4 Evaluation

In this section, we evaluate the performance of RobuSTM. We are specifically interested in showing that (1) it provides high throughput in good cases with little contention, (2) it provides fairness by guaranteeing progress of individual transactions, and (3) it tolerates crashed and non-terminating transactions.

We compare RobuSTM against four state-of-the-art STM implementations: TinySTM [6]; TinyETL, a C++ implementation of the encounter-time locking variant of TinySTM; TL2 [4], an STM implementation that uses commit-time locking; and NB STM [15], which combines efficient features of lock-based STM implementations with a non-blocking design, as our algorithm does. The NB STM implementation that we use is a port of the original SPARC implementation to the `x86` architecture.

For our evaluation, we use well-known micro-benchmarks and applications of the STAMP [2] benchmark suite. The *intset* micro-benchmarks perform queries and updates on integer sets implemented as *red-black tree* and *linked list*. We use the *bank* micro-benchmark to evaluate fairness: some threads perform money transfers (i.e., one withdrawal followed by a deposit) concurrently with long read-only transactions that compute the aggregated balance of all accounts. From the STAMP benchmark suite [2] we chose *Vacation*, *KMeans* and *Genome*. Vacation emulates a travel reservation system, reading and writing different tables that are implemented as red-black trees. KMeans clusters a set of points in parallel. Genome performs gene sequencing using hash sets and string search.

Our tests have been carried out on a dual-socket server with two Intel quad-cores (Intel XEON Clovertown, executing 64-bit Linux 2.6). We compiled all micro-benchmarks using the Dresden TM Compiler [3], which parses and transforms C/C++ transaction statements and redirects memory accesses to an STM.

4.1 Throughput for Well-Behaved Transactions

We first evaluate transaction throughput for lock-based and nonblocking STM implementations. There are no crashes or non-terminating transactions present.

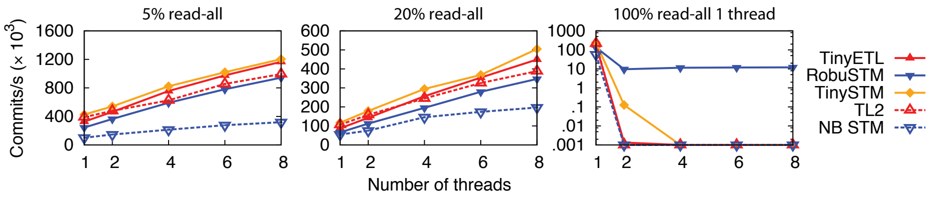

Fig. 6. Comparison lock-based vs. nonblocking STM (bank benchmark, 4096 accounts).

Figure 6 shows the *bank* benchmark with low load under different STM runtimes. The left and middle plots show throughput for both transfer and aggregate-balance transactions. The lock-based STMs perform significantly faster than NB STM because the chosen nonblocking STM still requires an indirection step in case of contention. These results show why we would like RobuSTM to perform as well as blocking STMs. RobuSTM has more runtime overhead than TinyETL and TinySTM but is on par with TL2. Figure 7 shows performance results for additional micro-benchmarks and STAMP applications. Results for NB STM are only presented for the red-black tree because it requires manual instrumentation and it is not supported by the STAMP distribution. These results are in line with the *bank* benchmark results, showing that TinySTM and TinyETL perform best, followed by RobuSTM, then TL2 and finally NB STM.

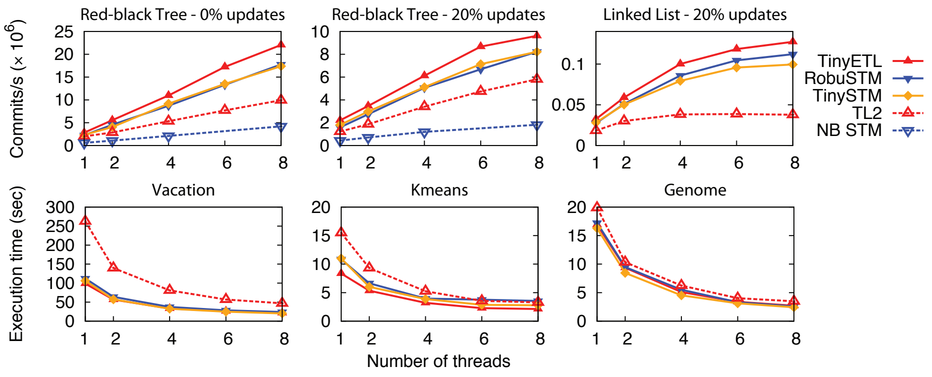

Fig. 7. Comparison of the performance of RobuSTM with micro-benchmarks (4096 initial elements) and STAMP applications (all with high contention).

The right plot of Figure 6 shows that the fairness that RobuSTM helps avoid starvation of the long aggregate-balance transactions with visible reads. In this plot, we only show the throughput of a single thread that is performing aggregate-balance (read-all) transactions. The guarantee for individual threads to make progress under MSM provides fairness for transactions that otherwise would not have a good chance to commit. Other STMs with invisible reads that simply abort upon conflict do not perform well because the read-only transaction will be continuously aborted.

4.2 Tolerating Crashes and Non-Terminating Transactions

We now evaluate transaction throughput in the presence of crashes or non-terminating transactions. Ill-behaved transactions are simulated by injecting faults at the end of a transaction that performed write operations (i.e., it holds locks). We inject thread crashes by raising a signal and simulate non-terminating transactions by entering an infinite loop. The infinite loop either performs no operations on shared memory or continuously executes STM operations (e.g., transactional loads).

Orthogonal to the robustness that RobuSTM offers for synchronization, applications must be tolerant against faults of its threads. During the setup of our experiments we discovered two major problems with the thread-based benchmarks. (1) Barriers must be tolerant to faulty threads that never reach the barrier because of a crash or non-terminating code. (2) The workload cannot be pre-partitioned to the initial number of threads. Instead, it must be assigned dynamically, e.g., in each loop iteration. Therefore, we chose only a selection of STAMP applications that could be easily adapted. Using RobuSTM, an adapted application with initially N threads can tolerate up to $N-1$ faults because even ill-behaved transactions prevent the thread from processing its work. Increasing the number of tolerated faults would require a change in the programming model, e.g., based on a thread pool, and is not in the scope of this paper.

Figure 8 shows the performance of RobuSTM compared to TinyETL, the most efficient STM in the previous measurements. In each plot of the figure,

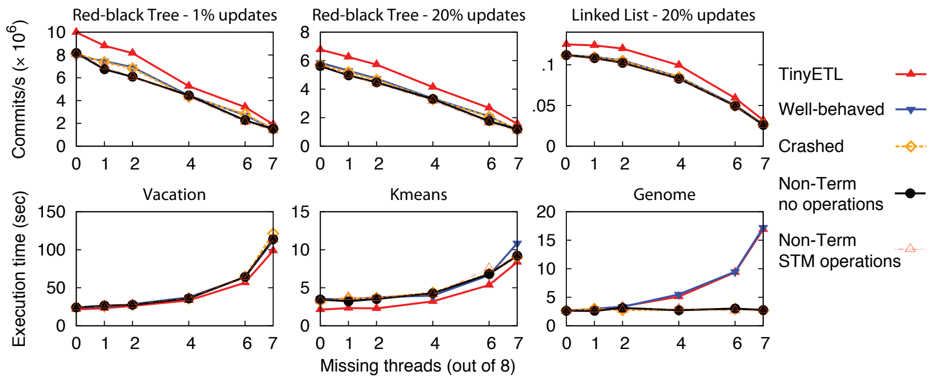

Fig. 8. Comparison of the performance for the benchmarks from Figure 7 with injected crashes and non-terminating transactions executing an infinite loop with or without STM operations.

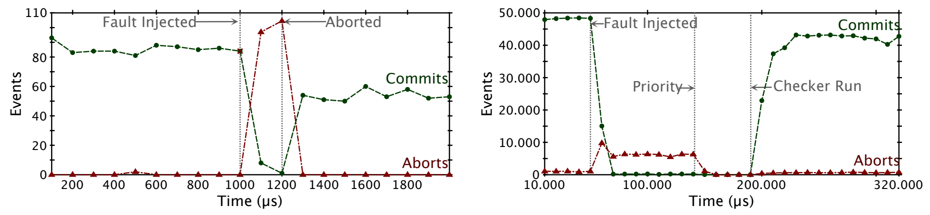

Fig. 9. Throughput for the red-black tree over time intervals under the presence of non-terminating transactions executing an infinite loop without (left) and with (right) STM operations. Vertical lines mark events of the fault-injected thread.

we show the performance when some threads are faulty in the baseline 8-thread run. TinyETL is a run where faulty threads are simply not started in the runs, and thus shows the baseline. "Well-behaved" is similar (only well-behaved transactions), but uses RobuSTM. The other three lines show the performance in the presence of transactions that are not well-behaved. Faults were injected as early as possible, except for Genome, where they were injected in the last phase and can only be compared to the 8-thread runs. The results show that RobuSTM can ensure progress for an increasing number of injected faults. In fact, it can even compete with the throughput of the "well-behaved" case for the considered benchmarks.

To illustrate how RobuSTM behaves when non-terminating transactions are present, Figure 9 shows the number of commits and aborts over periods of time for the red-black tree benchmark. In the left graph, the benchmark is executed with two threads and one transaction enters an infinite loop that does not call STM operations. The remaining thread runs into a conflict and aborts repeatedly until it enters the privileged priority level. It then is allowed to kill the non-terminating transaction in order to steal its locks. Afterwards, the throughput picks up to the level of a single threaded execution. The right graph shows a scenario taken from Figure 8 with eight threads and one transaction that enters an infinite loop with STM operations. All remaining seven threads abort after running into a conflict and eventually reach privileged priority. Because the non-terminating transaction detects that it has been aborted during its STM operations, it retries. Thus, it must be aborted multiple times until it gains privileged priority. While the non-terminating transaction executes privileged, other threads wait and check the quantum of the non-terminating transaction. After the transaction detects that it was aborted because its quantum expired, it will clone its thread to enter the checker run. The period between gaining privileged priority and entering the checker run is much longer than the allowed quantum because it includes the costly clone of the process. After the initialization of the checker is finished, all locks are released in the parent process and the other threads can continue.

The results show that despite several crashed or non-terminating threads, RobuSTM is able to maintain a good level of commit throughput, effectively shielding other threads from failed transactions. We tested injecting faults in other STM implementations to justify our design decisions. Lock-based designs that acquire locks at commit-time (e.g., TL2) seem promising towards tolerating crashes and non-terminating transactions. Problems arise when fairness is at stake because memory accesses cannot be easily made visible. For implementations with encounter-time locking that simply abort on conflict (e.g., TinyETL), transactions that are not well-behaved and own locks lead to deadlocks. To overcome deadlocks, lock stealing and external abort of transactions must be supported. This will allow to tolerate crashes but not non-terminating transactions as they might continuously retry. We found that none of further existing approaches for contention management (see Section 5) met our robustness requirements.

5 Related Work

Non-blocking concurrent algorithms (e.g., lock-free or wait-free) ensure progress of some or all remaining threads even if one thread stops making progress. While many early STMs where non-blocking, most of the recent implementations use blocking algorithms because of their simpler design and better performance. Recent work on non-blocking STM [16, 21] has shown that its performance can be substantially increased by applying techniques known from blocking STM implementations. This includes (1) timestamp-based conflict detection and (2) a reduced number of indirections while operating on transactional data by accessing memory in place in the common case. Depending on the algorithm, costly indirection is still required either during commit [16] or when stealing ownership records on conflict [21]. For the later, deflating the indirection is only possible after the original owner transaction moved to the abort state, but this might never happen for transactions that are not well-behaved.

Contention management was originally introduced to increase the throughput and avoiding possible livelocks (e.g., *Polite*, *Karma*, *Polka* [19, 18]). An interesting observation is to back off the losing transaction after a conflict to avoid encountering the same conflict immediately upon retry. Contention managers that aim to provide fairness between short and long-running transactions usually rely on prioritization. The priority can be derived from the time when a transaction started or the amount of work it has done so far [18, 9]. This helps long-running transactions reach their commit point but can delay short transactions extensively in case of high contention. Furthermore, crashed transactions will gain a high priority if it is based on the start time. An alternative is to derive the priority from the number of times the transaction has already been retried [19] and favor transactions with problems reaching their commit point.

In combination with priorities, simple mechanisms such as recency timestamps or liveness flags were introduced to determine the amount of time that contending transactions should back off. The goal is to increase the likelihood that a transaction that has already modified a memory location can commit (e.g., *Timestamp*, *Published Timestamp* [19, 18]). These mechanisms are also used by transactions to show that they are not crashed. However, this approach does not work for non-terminating transactions because they may well update the timestamp or flag forever. The length of potential contention intervals can be reduced if locks are not acquired before commit time [20]. This would allow us to tolerate non-terminating transactions because they never try to commit [10], but by detecting conflicts lazily one cannot ensure that a transaction will eventually manage to commit (it can be repeatedly forced to abort by concurrent transactions that commit updates to shared memory).

Contention managers can also try to ensure progress of individual transactions. In the initial proposal of the *Greedy* contention manager [9], which guarantees that every well-behaved transaction commits within a bounded amount of time, thread failures could prevent global progress (i.e., the property that at least some thread makes progress). This issue was solved by giving each transaction a bounded period of time during which it could not be aborted by other transactions [8]. If a correct transaction exceeds this time limit and is aborted, it can retry with a longer delay. This approach works for crash failures but not for non-terminating transactions because the delay can grow arbitrarily large if the transaction is retried infinitely often.

Fich et al. [7] proposed an algorithm that converts any obstruction-free algorithm [11] into a practically wait-free one. The idea is that, in a semi-synchronous system, it is impossible to determine if a thread has crashed by observing its executed steps, as a step can take a bounded but unknown amount of time to complete. Thus, it is not possible to know a priori how long to wait for a possibly crashed transaction. Instead, one has to wait for increasingly longer periods. To decide if a thread had indeed crashed after expiration of the waiting period, they observe the *instruction counter* of the thread used to track progress. This approach cannot be applied straightforwardly to STMs because transactions can contain loops or perform operations with variable durations, e.g., allocate memory, so we cannot automatically and efficiently determine the abstract linear instruction counter of a running transaction.

Guerraoui and Kapalka were the first to take non-terminating transactions explicitly into account [10]. Their result is that the strongest progress guarantee that can be ensured in asynchronous systems is global progress, which is analogous to lock freedom. Since thread crashes and non-terminating transactions are not detected but tolerated, one cannot give to a single transaction an exclusive execution right because the thread might gain the right and never release it. We show in this paper that relying on a different but yet practical system model (see Section 2) allows us to build robust STMs that avoid these limitations and work on current multicore systems.

6 Conclusion

Robustness of transactional memory has often been ignored in previous research as the main focus was on providing performance. Yet, robustness to software bugs and application failures is an important property if one wants to use transactional memory in large mission-critical or safety-critical systems.

In this paper, we have introduced the multicore system model (MSM) that is practical in the sense that it reflects the properties of today's multicore computers. We have shown that (1) it is possible to build a robust STM with performance comparable to that of non-robust state-of-the-art STMs, and (2) we can implement such an STM under MSM.

Our experimental evaluation indicates that robustness only has a small additional overhead in the good case (i.e., no or few ill-behaved transactions), and performance remains good even when there are crashed and non-terminating threads. We expect to further improve efficiency by tuning the configuration parameters at runtime. For RobuSTM, these are especially the number of retries (1) after which transactions switch to using visible reads and (2) after which they attempt to run as a privileged transaction. Previous work has shown this to be very beneficial in the case of other STM configuration parameters [6]. We also expect that pairing this work with operating-system scheduling [14] could enable interesting optimizations.

Acknowledgements. This work is supported in part by the European Commission FP7 VELOX project (ICT-216852).

References

1. M. Aguilera and M. Walfish. No time for asynchrony. In *HotOS'09: Proceedings of the 12th Workshop on Hot Topics in Operating Systems*. USENIX, 2009.
2. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.
3. D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys 2010*, 2010.

4. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006.
5. R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research, 2006.
6. P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.
7. F. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-Free Algorithms can be Practically Wait-Free. *Lecture Notes in Computer Science*, 2005.
8. R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *Workshop on Synchronization and Concurrency in Object-Oriented Languages*, 2005.
9. R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005.
10. R. Guerraoui and M. Kapalka. How Live Can a Transactional Memory Be? Technical report, EPFL, 2009.
11. M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software-transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing*. ACM, 2003.
12. T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-start: quick fault recovery in oracle. *SIGMOD Rec.*, 2001.
13. L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
14. W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, January 2010.
15. V. J. Marathe and M. Moir. Efficient nonblocking software transactional memory. Technical report, Department of Computer Science, University of Rochester, 2008.
16. V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.
17. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, 2006.
18. W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005.
19. W. Scherer III and M. Scott. Contention Management in Dynamic Software Transactional Memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, 2004.
20. M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2008.
21. F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009.