Dividing Transactional Memories by Zero

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

Abstract

Software transactional memory (STM) is a promising technique for writing concurrent programs. So far, most STM approaches have been experimentally evaluated with small-scale μbenchmarks. In this paper, we present several surprising results from implementing and experimenting with STMBench7 – a large scale benchmark for STMs. First, all STMs we used crashed, at some point or another, when running STMBench7. This was mainly due to memory management limitations. This means that, in practice, none of the tested STMs was truly unbounded and dynamic, which are the actual motivations for moving away from hardware transactional memories (HTM). Performance results we gathered also differ from previously published results. We found, for instance, that conflict detection and contention management have the biggest performance impact, way more than other aspects, like the choice of lock-based or obstruction-free implementation, as typically highlighted. Implementation of STMBench7 with various STMs also revealed several programming related issues such as the lack of support for external libraries and only partial support for object oriented features. These issues prove to be a major limitation when adapting STMs for production use.

Our work is by no means a bashing of prior work on STMs. All STMs we considered are very well designed and implemented. What we highlight here is that providing genuinely unbounded transactions is a hard and complicated task, but full of interesting technical and research problems. Solutions to these problems should be evaluated against large scale benchmarks, like STMBench7.

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

General Terms Measurement, Performance, Experimentation

Keywords Software transactional memories, Benchmarks

1. Introduction

Transactional memory (TM) systems are commonly regarded as a way of making concurrent programming usable by a wider community of programmers. TMs have first been designed in hardware [1]. Software transactional memories (STM) have been later proposed to overcome the bound on transaction sizes that is typical for hardware solutions [2] and to support *unbounded* and *dynamic* transactions, i.e., transactions with no bound on the number or sizes of objects accessed.

There have been many recent proposals for STMs [2, 3, 4, 5, 6]. All of them were validated using well-known data structures (linked lists, red-black trees, etc.). While these μbenchmarks can exhibit performance differences between various STMs to a certain extent, they have an important deficiency – they are too small in scale. The goal of STMs is to provide support for implementing real-world applications, such as big application and web servers, on emerging multi-core and multi-processor architectures. Current μbenchmarks do not provide sufficient insight about the behavior of STMs in applications with large transactions that access many, potentially complex, objects.

We recently proposed a more realistic benchmark for evaluating STM implementations – STMBench7 [7]. The main characteristic of STMBench7 is that it "stretches" the underlying TM system in ways various μbenchmarks do not. The main "stretcher" is the size of the data structure, which increases total memory requirements, transaction length and the number of objects accessed in transactions. STMBench7 was first implemented in Java with ASTM [8] and two levels of locking. We then ported STMBench7 to C++ and implemented it with four STMs: DSTM2[1] [6], RSTM [3], TL2 [4] and TinySTM [5].[2] The main reason for choosing these libraries was their free availability and their representation of various STM design choices. We could thus compare: managed (Java) against unmanaged (C/C++) runtime, lock-based against obstruction-free STMs and different conflict detection and contention management techniques. We present here the main conclusions that resulted from "stretching" these STM implementations with STMBench7.

Robustness *All* STMs we used had problems coping with the size of a big data structure. In short, they all crashed. This means that, in practice, none of these STMs were truly unbounded and dynamic. Some STMs incur too high space overheads and could not fit in the memory, others overflow internal data structures and either lead to incorrect executions or crashes. These issues might not be surprising given that memory management is particularly difficult in STM environment. First, STMs require more memory than regular programs as they have to maintain redo or undo logs. Also, deallocating memory in environments without garbage collector is more difficult as the reachability of memory blocks must be computed before deallocation. It should be mentioned that, although these libraries are not production quality, they were tested extensively and work without problems in a number of different μbenchmarks. Consequently, we did not expect to find them to crash, and that was not our objective.

[1] We were not able to make a fully functional version of STMBench7 with DSTM2 due to memory management related problems. We describe problems we encountered below.

[2] All STMBench7 versions are available for download at `http://lpd.epfl.ch/site/research/tmeval`.

It is interesting to contrast STM implementations with programs that use locking at this point. When locking is used, no memory is required for bookkeeping purposes and all memory blocks that are to be deleted are protected with appropriate locks, so there is no need to compute reachability before deleting them. Consequently, memory management is much simpler and STMBench7 locking implementations worked with standard memory allocation without any problems.

Performance STMBench7 also provides a performance test that stresses STMs differently than common μbenchmarks do. All μbenchmarks tend to measure lower level aspects of the design, like the overhead of each single interaction with the library or the amount of cache invalidation caused. This is due to the small transaction sizes used by μbenchmarks, which cause even very small overheads to have a significant relative impact. STMBench7's larger transactions are not impacted so significantly by these overheads, and the higher-level design issues can be compared. Approaches that have higher overheads on single object accesses might actually perform better with larger transactions, if they are able to avoid overheads of aborting transactions or performing too much work in transactions that are aborted later. Another important consequence of running large transactions is their ability to fully expose all superlinear performance overheads in the STM. The impact of superlinear overheads is much higher on programs that use larger transactions than on those that only use smaller ones.

Our experiments with STMBench7 led us to conclude that conflict detection and contention management have the biggest influence on performance. This contrasts previous conclusions, based on μbenchmark experiments, that identified low level overheads, such as the cost of accessing a single object, as the main performance factor [9]. We identified the conflict detection technique that has the highest performance for large transactions over a wide range of contention levels, unlike [3], in which μbenchmark experimental results did not favor any of the variants and proposed adaptive conflict detection. More specifically, in our experiments, techniques that detect write/write conflicts early and read/write conflicts late performed best. Both RSTM, with invisible reads, eager conflict detection and global commit counter heuristic switched on, and TinySTM use this kind of conflict detection. We also noticed an interesting relation between contention management and conflict detection. For example, Polka [10] contention manager works better than Greedy [23] with lazy conflict detection, but it is opposite with eager conflict detection. We were able to pinpoint the best performing contention management technique given a conflict detection choice. Our results are in contradiction with the previous study [10] based on typical μbenchmarks that favored Polka.

Application programming One of the unique features of STMBench7, when compared to other benchmarking approaches, is the use of (a) external libraries that do not support STMs and (b) a bigger subset of object-oriented features of the language (e.g. polymorphism). This serves as a sort of API (or usability) test for STMs, because it highlights some of the problems an STM might have when interfacing with external, non-transactional code. It also highlights the costs of using an STM in terms of either limited usability (limiting the use of external libraries) or additional programming effort (porting parts of external libraries to STM environment). For example, no word-based STMs we used fully support use of external libraries, which considerably increases the cost of using them. This kind of information might be useful for both software developers and compiler vendors.

To summarize, our experiments with STMBench7 revealed the following:

(a) *all* STMs we used crashed, in one way or another, mostly due to problems related to memory management

(b) the choice of appropriate policies (e.g. conflict detection technique and contention management) has bigger impact on performance than the choice of mechanisms (e.g. lock-based vs. obstruction-free implementation)

(c) some tested libraries have serious usability issues, mainly with object oriented features of the language and external libraries.

As we pointed out, mentioning various problems with particular STMs does not have a purpose of bashing these libraries, but merely of highlighting challenges underlying STMs.

The rest of the paper is laid out as follows. The next section provides some basic background information. The section describing some of the most interesting bugs discovered using STMBench7 follows. Next, we continue on to discuss some interesting performance results and follow with the section that highlights the biggest API limitations discovered. The last section summarizes our conclusions.

2. STM context

STMs can be word-based or object-based, depending on the granularity of memory accesses. Regarding the STMs we used, RSTM and DSTM2 are object-based, while TL2 and TinySTM are word-based. There are also two typical ways of implementing STMs – lock-based and obstruction-free. The main part of RSTM is obstruction-free,[3] while TL2 and TinySTM use locks. DSTM2 has both lock-based and obstruction-free variants.

No matter what the underlying details of a TM are, its main task is to *detect* conflicts among concurrent transactions and to *deal* with them. Deciding what to do when conflicts arise is known as *contention management* and is performed by a (conceptually) separate system component called a *contention manager*. A concept closely related to conflict detection is the one of *validation*. Validating a transaction consists of checking its read set for consistency. Figure 1 illustrates conflict detection and contention management with a simple example.

2.1 Conflict detection

Conflicts among concurrent transactions can be detected in a number of ways. In principle, the major factor is the point in the execution when read/write and write/write conflicts are detected. Both kinds of conflicts can be detected at different phases during the transaction execution.

RSTM supports four conflict detection approaches, each characterized by (a) the point in the execution when writers acquire objects – as soon as possible, called *eager*, or as late as possible, called *lazy* and (b) the visibility of readers – they can be *visible*, which allows writers to detect read/write conflicts or *invisible* [3]. RSTM also implements a global commit counter heuristic that reduces the cost of validation and also partially implements mixed invalidation [11].

TL2 and TinySTM are both timestamp-based TM systems that employ invisible reads. The difference in conflict detection is that writers lazily acquire locks while in TL2 and eagerly in TinySTM.

2.2 Contention management

Once the conflict is discovered, it is resolved using some form of contention management. A contention manager is a component that decides what a given transaction (*attacker*) should do in case of a

[3] RSTM has also a lock-based implementation, but it does not allow use of any external libraries, so we were not able to run STMBench7 with this RSTM variant.

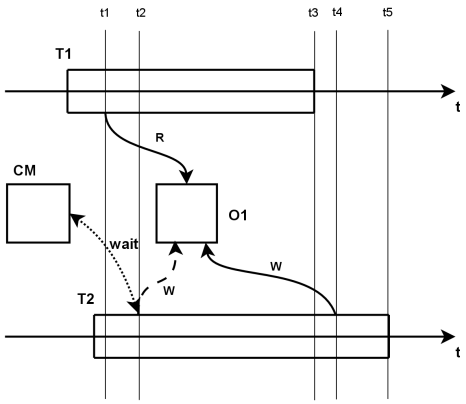

Figure 1. An example of read/write conflict at time `t2`. In this scenario, contention manager decides that `T2` should wait (depicted in dotted lines). After `T1` commits at `t3`, `T2` succeeds in writing at `t4` and commits at `t5`.

conflict with another transaction (*victim*). Possible outcomes are aborting the attacker, aborting the victim or forcing the attacker to wait for a while before trying again.

RSTM provides a plethora of contention managers, described in detail in [10, 12]. Here, we give a short description of the contention managers that are mentioned in the rest of this paper. *Polka* assigns to 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 backoff 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 always avoids livelocks among transactions. *Serializer* is very similar to Greedy except that it assigns a new timestamp to a transaction on every restart. TL2 and TinySTM have a very simple contention management technique – whenever there is a conflict, the attacker gets aborted and restarted. Also, TL2 makes the attacker wait for a while (exponential backoff) before restarting.

2.3 Benchmarking

The prevailing way of measuring the performance of STMs has been the use of μbenchmarks. The main goal of μbenchmarks is to test the mechanics of an STM itself and compare low-level details of different implementations. Simple operations on simple data structures of modest sizes serve this purpose well, as real application work does not mask TM implementation overheads. However, this also presents a danger for precisely the same reason, as some of the design choices that might look justified with μbenchmarks might actually lead to worse performance when more elaborated data structures and operations are used.

Some more realistic benchmarking approaches besides STMBench7 have also been considered. SPLASH-2 [13] is a suite of highly parallel applications that was designed for comparing different architectural aspects of shared-memory multiprocessor computer systems. In particular, it has been used as a benchmark for a number of HTM systems [14, 15, 16]. The main drawback of using SPLASH-2 for TM evaluations is the structure of SPLASH-2 programs. Threads in these programs use fine-grained synchronization during short periods of their execution and spend most of the time performing demanding calculations on data not accessed by other threads. While this provides higher performance, it is in contrast to expected TM usage patterns.

STAMP [17] is another, more recent benchmarking suite, which is better suited to TM evaluations than SPLASH-2, because it uses applications that spend most of the time executing transactional code and, thus, correspond to the expected TM usage patterns better. The main drawbacks of STAMP are the lack of (a) workloads with very long transactions and (b) use of complex data types, like strings or collections from standard libraries.

Benchmark	Data size	Name	Tx Size	Tx/s
μBench	128	Linked list	64	10^6
		Hash table	2	10^6
		RB tree	7	10^6
STMBench7	700,000	LT on	100,000	10^1
		LT off	large 10,000 avg $\sim$ 100	10^3

Table 1. Comparison of STMBench7 and μbenchmark average sizes. Data and transaction sizes are reported in numbers of objects. Sizes for typical μbenchmarks are given with the integer value range of 256. Size of STMBench7 transactions with long traversals on and off are given.

Category	Workload type		
	Read	Read-write	Write
Read-only ops	90	60	10
Update ops	10	40	90
Long Traversals	5		
Short traversals	40		
Short operations	45		
Structure mods	10		

Table 2. Default ratios for operation categories (in %)

2.4 STMBench7

STMBench7 [7] was specifically targeted at benchmarking STMs. Its data structure and operations, in large part inherited from the OO7 [18] benchmark for object-oriented databases, represent a workload typical for CAD/CAM/CASE software. This means that, although CAD/CAM/CASE applications are probably not characteristic of the future STM applications, STMBench7 workloads correspond to realistic, complex, object-oriented applications and as such represent very important target for STMs. Also, these workloads use transactions that are larger than in any other benchmark we know of, thus providing insight into STM behavior in unique scenarios.

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 static, read-dominated to dynamic write-dominated). The data structure used by STMBench7 is many orders of magnitude larger than in a typical μbenchmark. Also, its transactions are longer and access a larger number of objects. The difference between STMBench7 and typical μbenchmarks sizes is highlighted in Table 1. The size and variety of the STMBench7 transactions stress STMs in different ways than μbenchmarks do. STMBench7 can be, as experience shows very effectively, used as a crash, performance and API testing tool for STMs.

A run of STMBench7 consists of creating a randomized data structure and invoking a series of operations on it, each in a separate transaction. STMBench7 uses a tree-like data structure that is depicted in Figure 2. There are four main categories of STM-

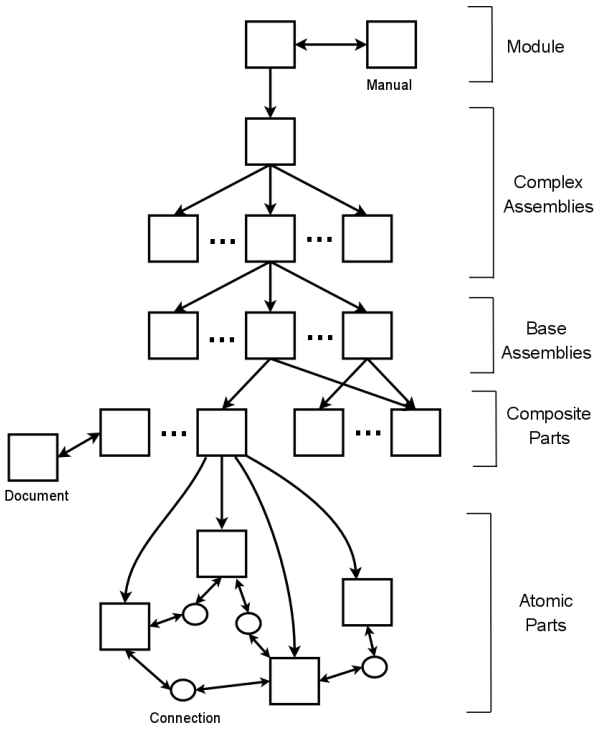

Figure 2. The data structure of STMBench7 consists of a single module that is connected to the tree consisting of six levels of complex assemblies. The last level of complex assemblies is connected to base assemblies. Base assemblies are connected to a number of composite parts that form a shared design library. Each composite part is connected to a private graph of atomic parts that are, in turn, connected to each other via connection objects. In addition, the module has a manual and each atomic part has a documentation object.

Bench7 operations. *Long traversals* are operations that access large parts of the data structure, typically all assemblies and atomic parts. *Short traversals* access much smaller number of objects, traversing the data structure along a random path. *Short operations* perform simple operations on a randomly selected object or its neighbours. *Structural modifications* change the data structure itself by creating or deleting objects or links among them. All structural modifications are performed in a way that prevents significant degeneration of the data structure (making parts of the structure disconnected from the root, for example). Operations are also classified into read-only and update operations, according to the type of accesses they perform. A distribution of operations that are invoked in a single run of the benchmark is based on the workload type selected, which can be read dominated, read/write and write dominated. In addition to selecting a desired workload type, long traversals can be included in the workload or not. This, in essence, leads to six different possibilities. A precise distribution of operations for different workloads is given in Table 2.

One peculiar characteristic of STMBench7 is the initialization of the data structure that is done in a single transaction. This transaction is not included in the measurements, but, due to the large size of the STMBench7 data structure, it turned out to be a big problem for some STMs and represents a major part of STMBench7 crash test.

STMBench7 with word-based STMs STMBench7 is inherently object-based. Its implementations also use standard language li-

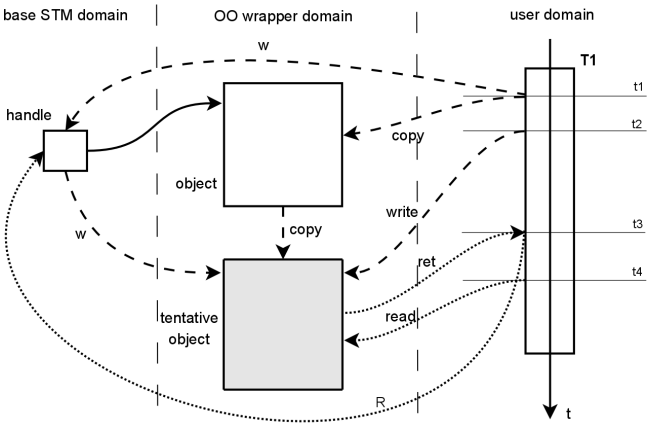

Figure 3. Example of an object oriented wrapper for word-based STMs. The figure shows an object handle that is accessed through an underlying STM, the actual object that is accessed through a wrapper and a transaction that uses the object. At time `t1` transaction `T1` opens the object for writing (shown in dashed lines). It makes a copy of the object and transactionally writes the new object's address to the handle. Later on, at time `t2` transaction `T1` writes to the copy of the object. At time `t3` the same transaction opens the object for reading (shown in dotted lines) by transactionally reading the value of the handle. This read returns the address of the object's copy and `T1` can later read data from the copy of the object (time `t4`).

braries (STL in C++ and standard Java library). To experiment with STMBench7 using word-based STMs, like TL2 and TinySTM, we implemented a thin object-oriented wrapper below STMBench7 (depicted in Figure 3). This wrapper can be thought of as an implementation of an object-based STM on top of a word-based one. The interface of this "composite" STM is very similar to the interface of RSTMv3, as that allowed us to easily reuse the bulk of benchmark's core code.[4]

The wrapper represents every object by its handle, which is actually a single memory word pointing to the current version of the object. The handle of the object is read and modified using underlying word-based STM, while the wrapper manipulates different object versions. Whenever a transaction opens an object for writing, a redo copy of the object is created and its address is speculatively written to the object's handle using underlying word-based STM. If the transaction commits, the redo version will become the current one, through mechanisms of underlying word-based STM, and if it aborts no change will be made. In addition, the wrapper maintains the log of all objects opened for writing and deletes newly allocated memory on aborts. Old object versions are deleted if the transaction commits. When a transaction opens an object for reading it reads the object handle using the underlying STM, which always returns the correct version of the pointer to the object, avoiding read-after-write hazards.

Besides object versioning, the wrapper also performs memory management functions, mostly rollbacking allocations on aborts and postponing deallocations of deleted objects until they are no longer reachable from other transactions. We adopted an approach similar to that of [19], McRT malloc [25] and RSTM [3]. Whenever a transaction allocates an object, the address of the object is logged. If the transaction aborts, it will delete all the objects that were allocated inside it using this log, effectively rollbacking memory allocation operations. When an object is deleted inside a transaction

[4] Initial implementation of STMBench7 for C++ was done with RSTM.

Crash	STM	Cause
Memory management	RSTM v2	16 MB/thread limit
	DSTM2	high overheads
	TinySTM	free on commit
Transaction size	RSTM v3	1 MB/object limit
	TL2 x86 0.9.0	internal overflow

Table 3. Summary of crash situations.

the actual delete is performed only if the transaction commits. Moreover, the object cannot be deleted immediately on commit, as some ongoing transactions might still hold references to it. Because of this, the object is deleted only when it is no longer reachable from any running transaction. The system maintains one counter per thread that counts the number of transactions executed in each thread in order to decide when it is safe to deallocate transactionally deleted objects.

Although our wrapper introduces additional overheads to word-based STMs, it is needed in order to use TL2 and TinySTM with C++ and non-transactional libraries. However, these overheads are comparable to those of RSTM v3 and this allowed us to better compare differences between lock-based and obstruction-free STMs, as they do not get masked by other implementation details.

2.5 Experimental settings

We performed all measurements on a 4 processor dual core AMD Opteron 8216 2.4 GHz 1024 KB cache machine with 8GB of RAM and Linux kernel version 2.6.16.37. This provided us with 8 cores to experiment on. Executables used in the experiments were created using `-O3` and `-DNDEBUG` compilation settings of GNU g++. All results were averaged over multiple runs, where the length and the number of runs were chosen to reduce variations in collected data. Experiments were performed with number of threads ranging from 1 to 24.

3. STMBench7 as a crash test

As we explain below, STMBench7 is highly effective in discovering STM problems that lead to program crashes. Two main reasons for STM crashing in our experiments were (a) memory management related problems and (b) the inability of STMs to cope with large transactions. Both of these reasons are rooted in the large size of STMBench7 data structure which stresses memory manager due to its high memory requirements and implies large transactions. The problems we encountered are summarized in Table 3.

3.1 Memory restrictions

As already pointed out, and somewhat to our surprise, we discovered that some STMs we considered could not cope with the memory size of STMbench7's data structure. In these cases, it was not possible to run STMBench7 at all before fixing the problem. Others had subtle problems with support for non-transactional data.

RSTM Implementation of STMBench7 in C++ immediately led to the problem of exhausting the available memory with RSTM [3] (version 2). RSTM v2 comes with a custom memory allocator, that has an arbitrary limit of 16 MB of allocatable memory per-thread. Given much higher memory requirements of STMBench7, it was not possible to even start it. Version 3 of RSTM has a flexible memory management module that allows the use of a wider range of memory allocators, including the standard `malloc`/`free` pair, which we could indeed use.

This bug, although quite easy to detect and fix, caused no problems with the μbenchmark tests, simply because none of them required allocating 16 MB of memory in a single thread.

DSTM2 DSTM2 [6] is written in Java, which has a built-in garbage collector that moves most of memory management related issues from programmers to the run-time system. This means that DSTM2, unlike RSTM, does not need any form of memory management in the library itself. DSTM2 is an experimental library that is very easy to extend and use with different STM algorithms in a transparent way from the programmers' perspective. To accomplish this, DSTM2 generates various classes during run-time (it has to generate a "transactional" version for every class of shared objects) and wraps every getter and setter method of these classes into separate objects.

The first problem we encountered was the exhaustion of memory that JVM uses to store class definitions. The reason for this was the way DSTM2 creates classes – one for every instance of the class, although a single version would suffice. After reporting this to the DSTM2 authors, we got the patched version of the library that did not exhibit the same problem. However, we were still not able to allocate the required data structure, this time due to the lack of heap memory. The problem persisted even when we increased the JVM heap size to 8 GB. After inspecting this problem further, we found out that, for every data member of the object, two wrappers, one for the getter and another one for the setter, were created. Both of these wrappers had a reference to a different instance of the `java.lang.reflect.Method` object. The size of the referenced `java.lang.reflect.Method` objects was an order of magnitude greater than the size of the rest of the data structure, which caused JVM to run out of heap memory. We were able to create a small patch that reused one `java.lang.reflect.Method` in all wrappers. Although this reduced memory consumption, it did not solve the problem either. DSTM2 still introduces big per-object overheads in its internal structures and this prevented STMBench7 initialization thread from finishing due to the JVM heap exhaustion. Fixing this problem turned out to require significant rework of the library, so we did not do it.

Because μbenchmarks do not create as many objects as STMBench7, these problems were overlooked. It is worth mentioning that the lock-based version of STMBench7 runs without any problems, with modest memory requirements (approximately 200 MB, which is far from exhausting JVM heap), as it does not incur any additional per-object overheads.

TinySTM TinySTM uses a simplified approach to memory management that is efficient in most cases and works correctly when all transactional object accesses are performed through provided load/store calls. The approach relies on a non-faulting load instruction, which can be simulated with appropriate signal handlers. Whenever a transactional deallocation is invoked, all locations in the block are transactionally written in order to prevent concurrent transactions from modifying them. Memory deallocations are postponed until the end of the current transaction and are performed only if the transaction commits.

Unfortunately, this memory management scheme caused crashes in our object-oriented wrapper. The wrapper caches object pointers during the transaction for performance reasons, and this allows concurrent transactions to access already deallocated objects. Illegal memory accesses are handled by TinySTM signal handlers, but they can cause memory corruption in rare cases when the same memory block gets allocated again by a different transaction. This typically happens in long transactions, when the old object handle gets used long after the object is deallocated. The problem most often manifests itself when copying large C strings, in cases when new contents of the memory block do not contain any zeroes that would stop the copying. Due to the resulting buffer overflows, heap sometimes gets corrupted, causing the next `malloc` or `free` to abort the program. Only in these cases we were able to detect the problem.

This problem is not obvious at all, as it seems that the non-faulting load instruction should prevent it. STMBench7 was instrumental in discovering it due to the large size of its transactions and data. Solving the problem required implementing object reachability algorithm in the wrapper's memory manager.

3.2 Transaction size

Some STMs had problems with the sizes of individual transactions, as opposed to the size of the whole data structure. Here the number of objects accessed matters, not their size, as transactions spend the fixed amount of bookkeeping memory for each accessed object.

RSTM RSTM v3 comes with a reworked memory management system that supports multiple possibilities for memory allocation. These include standard `malloc` implementations and the new version of RSTM's custom allocator that removes the 16 MB per-thread limit. However, there was still a problem that prevented the initialization thread from completing when using RSTM's custom allocator. We were able to discover and fix a pretty interesting bug that caused this. RSTM's allocator allocates memory in chunks that are at most 1 MB in size, which limits the maximum size of objects. As there are no objects larger than 1 MB in STMBench7, this did not present a problem when allocating STMBench7 specific objects. RSTM, however, internally uses arrays for storing sets of objects that transactions access and these arrays grow with the number of accessed objects. With large enough transactions, these arrays will require more than 1 MB of memory and will cause the program to crash. This practically limits the number of objects transactions can access.

The μbenchmarks did not reveal this problem, as they never access enough objects during a single transaction to cause overflows of allocated memory arrays. The long initializing transaction of STMBench7 crashed, not allowing us to use the custom memory allocator at all. To discover the problem was, again, harder than to fix it and a small patch to RSTM's custom allocator was enough.

TL2 The problem with the number of accessed objects in a single transaction also occurred with TL2 [4] x86 version 0.9.0 [17] (see Figure 4). TL2 stores addresses of all memory locations accessed for reading in a transaction-local linked list. This list is preallocated at the beginning of a transaction, for performance reasons. If a transaction reads more locations than can be stored in the list, it is aborted and restarted. Before the restart, the list is extended to accommodate a larger number of accessed locations. Due to a programming bug, transactions did not get aborted in the overflow case at all. Instead, reads that caused overflow were not accounted for at all and transactions were allowed to continue until commit time. At that point the validation algorithm would not take overflown reads in account and transactions could commit despite possible conflicts caused by missed reads. This violates transactional semantics and can even crash the program in some specific cases, which is how we actually noticed the problem.

In particular, when two transactions are trying to delete the same object, and the existing conflict passes undetected, both transactions will mark the same object for subsequent deletion, which will result in double delete of a memory block. Most of the time, TL2's signal handlers, that are used for emulating non-faulting load instruction on the x86 architecture, will catch the resulting segmentation fault and mask the problem. In some rare cases, however, the second free of the object causes subsequent `malloc` or `free` to abort the program, due to heap corruption. Only in these cases does the program crash and that is where we were able to notice the bug.

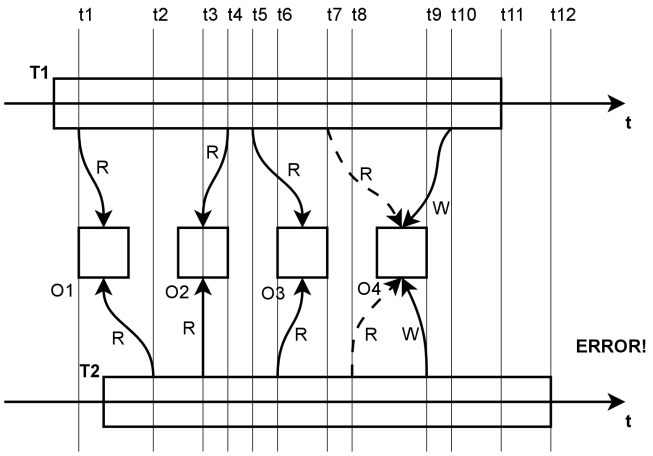

Figure 4. TL2 x86 version 0.9.0 read list overflow problem example, with the read set size of 3. Overflows of read set are depicted in dashed lines. Although this example scenario should result in a conflict, it does not because the read of the object `O4` is missing from `T2`'s read set.

A newer version of TL2 x86 fixed this problem by appropriately handling overflows. This was also not a very hard problem to solve, once it was discovered and STMBench7 helped us with that.[5]

4. STMBench7 as a performance test

We noticed that the biggest influence on the throughput of STMBench7 experiments comes from the combination of the conflict detection technique and the contention manager used in the underlying STM. Although other issues, like locking vs. non-locking implementation, also influence the overall performance, they do not impact it so significantly.

4.1 Locking vs. obstruction freedom

It was noticed previously [9, 24] that locking STMs have a performance edge over obstruction-free ones. We ran experiments with TL2 and TinySTM which are both lock-based STMs and got the confirmation of this. However, this turned out not to be the STM characteristic with the greatest influence on performance. Figure 5 shows that, with comparable conflict detection approaches, locking outperforms obstruction freedom, as TinySTM performs better than the RSTM variant using invisible reads with eager acquire and global commit counter heuristic. It also shows that the obstruction-free strategies can perform better than suboptimal locking ones, as the best RSTM variant outperforms TL2. This RSTM variant outperforms TL2 mostly because of a superior conflict detection approach, which uses early detection of write/write conflicts.

4.2 Towards the ideal conflict detection approach

We observed best throughputs when using conflict detection approaches that have the same characteristics – they are able to avoid unnecessary work and they avoid aborting potentially conflicting transactions as much as possible. The best performing RSTM conflict detection approach is invisible reads with eager acquire and global commit counter heuristic. This is precisely because this approach detects write/write conflicts early, so no further work is performed in vain in that case, but tries to detect read/write conflicts as late as possible, thus allowing more parallelism than techniques

[5] In this particular case, the authors of the library were able to find and resolve the problem on their own, prior to our report.

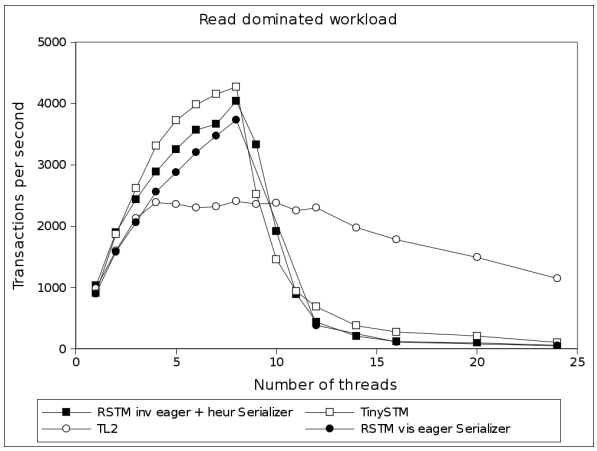

Figure 5. Comparison of different conflict detection approaches.

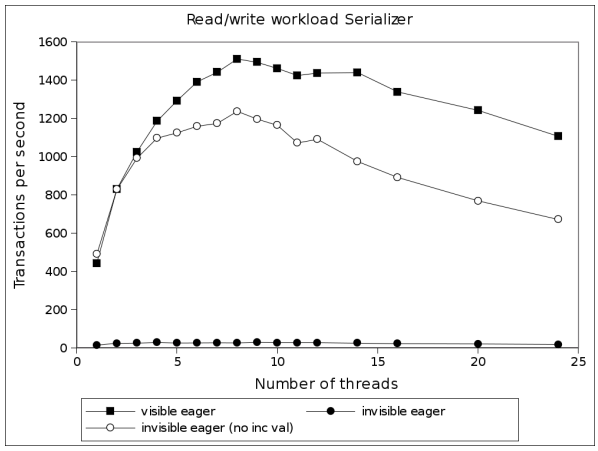

Figure 6. Incremental validation cost (RSTM).

using visible reads do. This is also the main reason why TinySTM outperforms TL2 drastically.

It can be conjectured that another step towards the ideal conflict detection approach would be to allow for even more parallelism between concurrent transactions. In terms of existing conflict detection approaches, both TinySTM and the RSTM variant that uses invisible reads with eager acquire and global commit counter heuristic could be changed to allow progress of both conflicting transactions even after the read/write conflict is detected, until it is certain that one of the transactions cannot commit. This change would let both transactions commit (and avoid wasting work of aborted transaction) in cases when the transaction that performed the conflicting read commits first. Having multiple versions of the same object, as suggested in [20] and [21], could further help as this would allow for more transactions to commit.

Visible reads It is worth mentioning that the visible reads approaches, which are commonly considered to be low-performing, have quite good performance in our experiments. In STMBench7 experiments, the RSTM variant that uses visible reads with eager acquire is second best performing RSTM variant and third overall (Figure 5).

Some prior experimental results [3] showed that, for μbenchmarks, invisible reads, even without global commit counter heuristic,[6] in almost all cases outperform visible reads. This was attributed to lower cache invalidation rate of the basic invisible reads approaches. However, in our experiments with RSTM, the quadratic cost of incremental validation became overwhelming for long transactions and basic invisible reads always have considerably lower performance. One extreme is the case with long traversals switched on, when runs with basic invisible reads do not finish a single long traversal in minutes. Example measurements with long traversals switched off are given in Figure 6. The difference is still considerable, even up to two orders of magnitude. To make sure that the reason for the low performance of basic invisible reads is incremental validation, we turned it off and repeated the measurements. As conveyed in Figure 6, the performance of basic invisible reads without incremental validation comes close to the performance of visible reads.

It might be quite surprising that basic invisible reads techniques, even without incremental validation, do not outperform techniques using visible reads. The reason is that RSTM visible reads with eager acquire variant comes closer to the best performing conflict detection techniques in terms of policy, as it detects doomed transactions early on and thus prevents loss of work. However, it detects read/write conflicts too early and that is why it does perform worse than invisible eager variant that uses global commit counter heuristic.

[6] We will call them basic invisible reads.

4.3 Towards the ideal contention manager

Similar observations that hold for conflict detection techniques can be applied to contention management – contention managers that work best (a) avoid aborting transactions and (b) if aborting is necessary they try to abort the transaction that performed less work. Although Polka [10] satisfies the second property, it aborts the victim after waiting for a certain time period and thus does not satisfy the first one. Two best performing contention managers are Serializer [22] and Greedy [23]. These contention managers use the elapsed time to measure the amount of work a transaction performed and always prioritize the one that performed more work. This means that "younger" transactions never abort "older" ones, but wait instead. On the other hand, "older" transactions always abort "younger" ones and are able to progress.

One interesting example of contention management is that of TinySTM. In locking STMs, especially in the ones with eager acquire like TinySTM, it is not easy to abort the victim as the changes performed to memory locations are unknown. This is why locking strategies usually abort the attacker or make it wait for a while and retry. TinySTM does not wait at all – on every conflict it aborts the attacker. It turns out that this approach works quite well, as conveyed by Figure 5, although it causes a huge number of aborts. A possible explanation for this is that in STMBench7 most operations access elements in roughly the same order, so transactions that start sooner will arrive at the same point in the data structure sooner. This represents a good approximation of the amount of work performed.

4.4 Conflict detection and contention management

During our experiments we noticed the strong relation between conflict detection techniques and contention management. Figure 7 shows that RSTM with the Polka [10] contention manager performs better when using lazy than eager conflict detection. On the other hand, Greedy works better with the eager approach. Similar observations can be made for other contention managers as well. This shows that comparing different contention managers while fixing a single conflict detection approach, or vice-versa, might not give the most representative results. Another way of looking at this is that the construction of a new contention manager requires prior knowledge of the conflict detection technique that will underlay it.

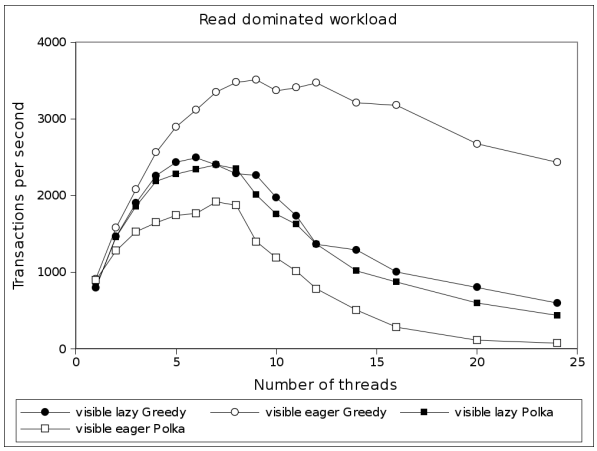

Figure 7. Relation between conflict detection techniques and contention management (RSTM).

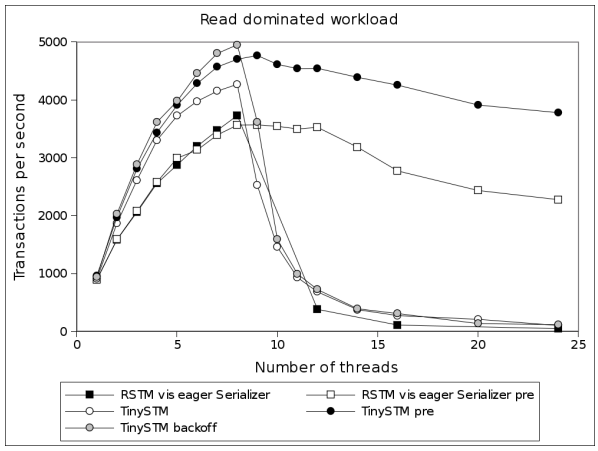

Figure 8. Preemptive vs non-preemptive implementations at high concurrency levels.

4.5 High concurrency levels

Running experiments with more threads than actual CPU cores results in unexpectedly sharp performance degradation in a number of different cases (Figure 8). An STM cannot further improve performance in these situations, as there is simply no more processing resources available in the computer system, but it should degrade performance gracefully. This is especially important once the STM applications reach common users, as it cannot be expected, or it might even be impossible, for the users to fine-tune all running applications according to the number of available cores.

The main reason for this significant performance impact is the busy waiting performed by waiting threads. Busy waiting wastes processors cycles, but even worse, it makes logically blocked threads look busy to the underlying scheduler. Because of this, a thread scheduling algorithm will assign cores to those threads, possibly preempting the ones that can actually make some progress. This will slow down the overall system progress even further. The problems with busy waiting are not noticeable with short transactions, because busy waiting ends as soon as the transaction can progress further, i.e., when the conflicting transaction finishes. When transactions are short, this period is short as well and busy waiting does not waste as much processor time as with longer transactions. It is even beneficial, as it reduces the "reaction" time of waiting threads by eliminating the overhead of thread switching.

We first noticed this problem with RSTM using the Serializer contention manager [22]. To test whether the busy waiting was the cause of the performance degradation, we replaced it with the simplest form of real waiting by invoking `yield()` whenever the contention manager decides to block the current transaction. As Figure 8 demonstrates, this solved the problem.

The similar performance degradation occurred with TinySTM and the reasons were pretty much the same – by aborting and restarting transactions immediately, other threads are prevented from using available CPUs. The simplest solution to the problem was the same as above – before restarting, an aborted transaction would first invoke `yield`. Interestingly enough, simple `yield`ing slightly improved performance of TinySTM, even when the number of threads was lower than the number of available CPUs. We attributed this to short backoffs a call to `yield` incurs even when no thread switch happens. These backoffs reduce contention, and, consequently, the number of aborts. To verify our assumption, we implemented a very simple busy waiting backoff mechanism in TinySTM. As conveyed by Figure 8, our assumption was correct. This is similar to the conclusion of [26], which states that expensive action on abort may act as a backoff and reduce contention, which results in improved performance.

5. STMBench7 as an API test

STMBench7 effectively highlights the problems that might be encountered by software developers or compiler vendors that start using a particular STM in production. The problems we identified fall mainly into two categories – lack of support for external libraries and lack of complete support for object oriented languages. For a discussion of different set of API problems of library based STM approach, the reader is referred to [27].

5.1 External libraries

Although STMBench7 uses only character strings and basic containers from standard libraries, this was enough to prevent the full utilization of some STMs. For example, neither TL2 nor TinySTM support the integration with any external library as they operate on single memory words. Because it was not possible to implement STMBench7 with a fully word-based approach without rewriting big parts of the standard library, we had to work around this limitation and we implemented a thin object oriented layer on top of these libraries, as mentioned above. We encountered similar problems with RSTM and its support for privatization techniques. Due to this limitation, we were not able to use privatization in STMBench7. This did not cause any major problems for us as STMBench7 is fully transactional.

Another problem that prevented the use of STL classes with RSTM was RSTM's implementation of `STLAllocator`. This allocator is suitable for use with fully transactional implementation of STL (which does not exist), as it replaces standard allocation and deallocation routines with their transactional equivalents. When a transaction that later aborts creates an object of a standard, non-transactional STL class using RSTM allocator, the object's memory gets deleted twice on abort – once by the object's destructor and once by a transactional abort rollback. This causes program crashes and, effectively, prevents the programmer from using objects of standard STL classes. We had to replace transactional versions of memory allocation routines with non-transactional ones. More generally, this led us to the conclusion that STMs that provide support for non-transactional versions of external classes need to provide both transactional and non-transactional memory allocation mechanisms.

5.2 Object-oriented features

We also ran into some less severe API-related problems that might make the life of implementers harder. Most of them were pretty easy to fix, but required a deeper knowledge of internal library details. An example of these issues is the lack of support for polymorphism in RSTM's smart pointer implementation. None of the μbenchmarks revealed this as all of them use simplistic class hierarchies which do not require polymorphism at all. For example, the most common μbenchmarks use classes only to represent container objects (trees and lists) and their nodes. On the other hand STMBench7 uses inheritance and polymorphism to represent `Assembly` class, with its two subclasses – `ComplexAssembly` and `BaseAssembly`. Every `ComplexAssembly` object holds its child `Assembly` instances, which could either be `ComplexAssembly` or `BaseAssembly`, in `sets`. It is important that the STM supports representation of references to child classes as references to parent classes in order for this to work. Once we discovered the problem it was not that difficult to patch the library and solve it.

Another issue is related to the fact that TinySTM is not oriented towards object-oriented programming and its memory manager does not invoke destructors of memory blocks when they are deallocated. Instead it just `free`s the memory block. This results in problems with C++ code that relies on destructors for correct functioning. The simplest solution in programs that only use objects, like STMBench7, is to replace calls to `free` with calls to `delete`, which will invoke the appropriate destructor. However, it is not completely clear how to combine deallocation of objects and plain memory blocks in a more general way.

6. Concluding Remarks

Although the ultimate goal of STMs is to help develop big programs running on thousands of processors, one might think that the large-scale scenarios of STMBench7 are not of interest while developing a new STM right now.[7] However, observations made on small-scale tests do not translate well to large-scale, thus testing of larger-scale systems is necessary. Similarly, one might feel that it is not yet the time to address usability issues of STMs, as they are all still in an experimental phase, or that these problems could be dealt with at different levels (e.g. in compilers) more elegantly. We, however, think that if the STM technology is to be used by a wider population, the cost of introducing it should be as small as possible. It might not really be acceptable to rewrite whole standard libraries of languages like C++ or Java, for example, like some word-based STMs require.[8]

To summarize, this paper presents some of our conclusions gathered during the implementation and experiments with STMBench7 using different STMs. In short, STMBench7 can, very efficiently, be used to test whether STMs are dynamic and unbounded in practice. Strangely enough, all STMs we used turned out not to be dynamic and unbounded, for various reasons that are mostly related to memory management. Another important STMBench7 property is that it can be used to compare the performance of different *policies* as opposed to comparing *mechanisms*. This is in contrast to most commonly used μbenchmarks. Finally, STMBench7 can also be used to assess the costs of adopting STM solutions in terms of usability and changes to external libraries, which might be a very valuable information when adopting a particular STM for use.

[7] After all, TM technology is in its infancy and most libraries are aimed at exploring different solutions to the problem, not at production use.

[8] And even if this is to be done, it is helpful to know what the real limitations of current approaches are and where can the compiler help.

References

[1] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[2] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software transactional memory for dynamic-sized data structures. In *22nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2003.

[3] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing (Transact)*. Jun 2006.

[4] O. Shalev D. Dice and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.

[5] Torvald Riegel Pascal Felber and Christof Fetzer. Dynamic performance tuning of word-based software transactional memory. In *The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Feb 2008.

[6] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 253–262, 2006.

[7] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference (EuroSys)*, pages 315–324. ACM, Mar 2007.

[8] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, Cracow, Poland, Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[9] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (Transact)*. Jun 2006.

[10] III William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 240–248, New York, NY, USA, 2005. ACM.

[11] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, Sep 2006.

[12] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, St. John's, NL, Canada, Jul 2004.

[13] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–38, June 1995.

[14] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 347–358, New York, NY, USA, 2006. ACM.

[15] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle

Olukotun. A scalable, non-blocking approach to transactional memory. In *13th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2007.

[16] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 254–265. Feb 2006.

[17] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. *SIGARCH Comput. Archit. News*, 35(2):69–80, 2007.

[18] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the 9th annual conference on Object-oriented programming systems, language, and applications (OOPSLA)*, pages 414–426, New York, NY, USA, 1994.

[19] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

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

[21] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (Transact)*. Jun 2006.

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

[23] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 258–264, New York, NY, USA, Jul 2005.

[24] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[25] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. Proceedings of the 5th international symposium on Memory management (ISMM), pages 74–83, Ottawa, Ontario, Canada, 2006

[26] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari and Emmett Witchel. MetaTM/TxLinux: transactional memory for an operating system. Proceedings of the 34th annual international symposium on Computer architecture (ISCA), pages 92–103, San Diego, California, USA, 2007

[27] Capabilities and Limitations of Library-Based Software Transactional Memory in C++. Dalessandro, Luke and Marathe, Virendra J. and Spear, Michael F. and Scott, Michael L. Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing (Transact), Portland, Oregon, USA, Aug 2007