Why STM can be more than a Research Toy

Aleksandar Dragojević[1] Pascal Felber[2] Vincent Gramoli[1 2] Rachid Guerraoui[1]
[1]EPFL, Switzerland
[2]University of Neuchâtel, Switzerland

Abstract

Software Transactional Memory (STM) promises to simplify concurrent programming without requiring specific hardware support. Yet, STM's credibility lies on the extent to which it can leverage multi-cores to outperform sequential code. A recent CACM paper [8] questioned however the ability of STMs to provide good performance and suggested their confinement to a research toy.

This paper revisits those conclusions through the most to date extensive comparison of STM performance to sequential code. We evaluate a state-of-the-art STM system, SwissTM, on a wide range of benchmarks and two different multicore systems. We dissect the inherent costs of synchronization as well as the overheads of compiler instrumentation and transparent privatization.

Our results show that an STM with manually instrumented benchmarks and explicit privatization outperforms sequential code by up to 29 times on SPARC with 64 concurrent threads and by up to 9 times on x86 with 16 concurrent threads. Indeed the overheads of compiler instrumentation and transparent privatization are substantial, yet they do not prevent STM from generally outperforming sequential code.

Keywords Software Transactional Memory, Performance

1. Introduction

While multicore architectures are becoming the norm in recent and upcoming CPUs, concurrent programming remains a difficult task. The transactional memory (TM) paradigm simplifies parallel programming by enabling the programmers to focus on high-level synchronization concepts (i.e., atomic blocks of code) while ignoring the low-level implementation details.

Hardware transactional memory (HTM) has already shown promising results for leveraging parallelism [39]. However, HTMs are so far limited as they can only handle transactions of limited size [10, 26], or require some system events or CPU instructions to be executed outside transactions [10, 32]. While there have certainly been attempts to address these issues (e.g., [4, 6, 34]), TM systems that are fully implemented in hardware are unlikely to become commercially available in the near future. It is more likely that future deployed TMs will be hybrid TMs that will contain a software and a hardware component.

Software Transactional Memory (STM) [24, 42] avoids limitations of HTM by implementing TM functionality fully in software, and is already freely available (e.g., [3,11,14,18, 23,35]). Yet, STMs introduce noticeable runtime overheads:

1. **Synchronization costs.** Each read (or write) of a memory location from inside a transaction is performed by a call to an STM routine for reading (or writing) data. With sequential code, these accesses are performed by a single CPU instruction. STM read and write routines are significantly more expensive than corresponding CPU instructions as they, typically, have to maintain bookkeeping data about every access. Most STMs check for conflicts with other concurrent transactions, log the access, and in case of a write, log the current (or old) value of the data, in addition to reading or writing the accessed memory location. Some of these operations use expensive synchronization instructions and access shared metadata, which further increases their costs. All of this reduces single-threaded performance when compared to sequential code.
2. **Compiler over-instrumentation.** To use an STM, programmers need to insert STM calls for starting and ending transactions in their code and replace all memory accesses from inside transactions by STM calls for reading and writing memory locations. This process, called *instrumentation*, can be *manual*, when the programmers manually replace all memory references with STM calls, or it can be performed by an STM *compiler*. With a compiler, programmers only need to specify which sequences of statements have to be executed atomically, by enclosing them in transactional blocks. The compiler generates code that invokes appropriate STM read/write calls. While using an STM compiler significantly reduces programming complexity, it can degrade performance of resulting programs (when compared to manual instrumentation) due to over-instrumentation [8, 15, 47]. Basically, the compiler cannot precisely determine which instructions indeed access shared data and hence has to instrument the code conservatively. This results in unnecessary

Model	Instrumentation	Privatization
STM-ME	manual	explicit
STM-CE	compiler	explicit
STM-MT	manual	transparent
STM-CT	compiler	transparent

Table 1. STM Programming models

calls to STM functions, reducing the performance of the resulting code.

3. **Transparent privatization.** Making certain shared data private to a certain thread is known as *privatization*. Privatization is typically used to allow non-transactional accesses to some data, either to improve performance by avoiding costs of STM calls when accessing private data or to support legacy code. Using privatization with unmodified STM algorithms (that use invisible reads) can result in various race conditions [44]. There are two different approaches to avoiding such race conditions: (1) a programmer marks transactions that privatize data, so the STM can safely privatize data only for these transactions or (2) the STM ensures that all transactions safely privatize data. We call the first approach *explicit* and the second *transparent*. Explicit privatization places additional burden on the programmer, while transparent privatization incurs runtime overheads that can be high [47]. In particular, with explicit privatization no transaction pays any additional cost if it does not use privatization, while with transparent privatization all transactions are impacted. This cost can be high, especially in cases when no data is actually being privatized.

Several research papers have conveyed the scalability of STM with the increasing number of threads on various benchmarks e.g., [2, 3, 5, 7, 11–14, 22, 25, 27, 29, 30, 33, 35, 36, 38, 40, 43, 46]. Most of this work, however, does not compare STM to sequential code, thus ignoring the fundamental question of whether STM can be a viable option for actually speeding up the execution of applications.

Two notable exceptions are [7] and [8]. In [7] STM is shown to outperform sequential code in most STAMP benchmarks, but only using a hardware simulator. Recently, [8] shows that with real hardware STM performs worse than sequential code, and argues that STM is only a "research toy". These findings are based on experiments with a subset of the STAMP benchmark suite with specific configurations and one micro-benchmark, all using up to 8 threads.

The goal of our work is to compare STM performance to sequential code using (1) a larger set of benchmarks and (2) a real hardware that supports higher levels of concurrency. To this end, we experimented with a state-of-the-art STM algorithm, SwissTM [14], running three different STMBench7 [21] workloads, all ten workloads of the STAMP (0.9.10) benchmark suite and four micro-benchmarks, evaluating STM performance on both large and small scale workloads. We also consider two hardware platforms—a Sun Microsystems UltraSPARC T2 CPU machine (referred to as *SPARC* in the remainder of the text) supporting 64 hardware threads and a 4 quad-core AMD Opteron x86 CPU machine (referred to as *x86* in the remainder of the text) supporting 16 hardware threads. This constitutes the most to date performance comparison of STM to sequential code both in terms of used benchmarks and hardware architectures. The goal is to really determine whether current state-of-the-art STMs can outperform sequential code and, thus, promise to speed up actual real-world code in the near future.

To be exhaustive we consider all combinations of privatization and compiler support for STM (summarized in Table 1). Our results (summarized in Table 2) show that STM-ME outperforms sequential code in all the benchmarks on both hardware configurations, except high contention write-dominated STMBench7 workload on x86 (1 out of 17 workloads); by up to 29 times on SPARC with 64 concurrent threads and by up to 9 times on x86 with 16 concurrent threads. Compiler over-instrumentation does reduce STM performance, but does not impact its scalability. STM-CE outperforms sequential code in all STAMP benchmarks with high contention and in all but one micro-benchmark (1 out of 14 workloads); by up to 9 times with 16 concurrent threads on x86. Support for transparent privatization impacts STM scalability and performance more significantly, but STM-MT still outperforms sequential code in all benchmarks on SPARC and in all but two high contention STMBench7 workloads, one high contention STAMP benchmark and one micro-benchmark on x86 (3 out of 17 workloads). STM-MT outperforms sequential code by up to 23 times on SPARC with 64 concurrent threads and 5 times on x86 with 16 threads. Even when both transparent privatization and compiler instrumentation are used (and even though we used relatively standard techniques for both), STM still performs well, outperforming sequential code in all but two high contention STAMP benchmarks and one micro-benchmark (3 out of 14 workloads), by up to 5 times with 16 concurrent threads on x86.

To summarize, our experiments show that STM indeed outperforms sequential code in most configurations and benchmarks, offering already now a viable paradigm for concurrent programming. These results are important as they support initial hopes about the good performance of STM, and motivate further research in the field.

Our results contradict these of [8] and we believe that reasons for this are three-fold: (1) STAMP workloads used in [8] present higher contention than the default STAMP workloads, (2) we use different hardware configurations, and (3) we used SwissTM [14], which has higher performance than TL2 which was used in [8].

Indeed there are various programming issues with the use of STM (e.g., ensuring weak or strong atomicity, se-

Speedup		STM-ME			STM-CE			STM-MT			STM-CT		
Hardware	Hw threads	avg	min	max	avg	min	max	avg	min	max	avg	min	max
SPARC	64	9.1	1.4	29.7	-	-	-	5.6	1.2	23.6	-	-	-
x86	16	3.4	0.54	9.4	3.1	0.8	9.3	1.8	0.34	5.2	1.7	0.5	5.3

Table 2. Summary of STM speedup over sequential code.

mantics of privatization, support for legacy binary code, etc). However, the alternative concurrency programming approaches like fine-grained locking or lock-free techniques are not easier to use. Such comparisons have been discussed in [18, 19, 22, 24, 41] and are outside of the scope of this paper.

The rest of the paper is laid out as follows. The next section details our evaluation settings. The following four sections present and discuss the experimental results for all four variants of SwissTM. The final section concludes. In the appendix we provide data from experiments with other state-of-the-art STMs that further support our main conclusions.

2. Evaluation settings

In this section, we overview SwissTM implementation, as well as the benchmarks and hardware we used for our experimental evaluation.

2.1 SwissTM

Synchronization algorithm. SwissTM [14] is a word-based STM that uses a variant of two-phase locking for concurrency control. It uses invisible (optimistic) reads and relies on a time-based scheme to speed up read-set validation, similarly to [11, 35]. By using lazy read/write conflict detection SwissTM reduces the number of unnecessary aborts due to false read/write conflicts. By detecting write/write conflicts eagerly, it avoids wasted work in transactions that are almost certain to abort. This conflict detection scheme is known as *mixed invalidation* [45]. SwissTM uses deferred updates. The contention management scheme underlying SwissTM aborts conflicting transaction that performed less work by using a shared counter to establish a total order among transactions, similarly to Greedy [20], but avoids updates to the shared counter for short transactions, resulting in a two-phase contention management scheme. This design was carefully chosen to provide good performance across a wide range of mixed workloads, as demonstrated in [14].

Privatization. We implemented privatization support in SwissTM using a simple validation barriers scheme described in [44]. To ensure safe privatization, each thread, after committing a transaction T, waits for all other concurrent transactions to detect the changes to shared data performed by T. Basically, the thread waits for all concurrent threads to commit, abort or validate before executing application code after the transaction.

```
foo() {
    atomic {
        x = 1;
        foo_1();
        ...
    }
}
foo_1() {
    y = 1;
    foo_2();
    ...
}
foo_2() {
    ...
}
```

Figure 1. An example of an atomic code block.

Compiler instrumentation. We used Intel's C/C++ STM compiler [3, 33] for generating compiler instrumented benchmarks.[1] The compiler simplifies the job of a programmer who only has to mark atomic blocks of code. For example, in Figure 1, the programmer only marks the code in function `foo` as transactional, and the compiler instruments the code in `foo`, but also in invoked functions `foo_1` and `foo_2`.

2.2 Benchmarks

STMBench7. STMBench7 [21] is a synthetic STM benchmark that models realistic large-scale CAD/CAM/CASE workloads. STMBench7 workloads consist of a large number of different operations (from very short, read-only operations to very long ones that modify large parts of the data structure). Based on the number of operations that modify certain data, STMBench7 defines three different workloads, with different amount of contention: read-dominated (10% write operations), read/write (60% write operations) and write-dominated (90% write operations). The main characteristics of STMBench7 is that it uses a data structure and transactions that are larger than in other typical STM benchmarks and is, thus, very challenging for STM implementations.

STAMP. STAMP [7] is an STM benchmark suite that consists of 8 different applications representative of real-world workloads. We give a short description of STAMP applica-

[1] Intel's C/C++ STM compiler only generates x86 code thus we were not able to use it for our experiments on SPARC.

Benchmark	Description
bayes	bayesian networks structure learning
genome	gene sequencing
intruder	network intrusion detection
kmeans	partition-based clustering
labyrinth	shortest-distance maze routing
ssca2	efficient graph construction
vacation	travel reservation system emulation
yada	Delaunay mesh refinement

Table 3. STAMP applications

Workload	Parameters
bayes	`-v32 -r4096 -n10 -p40 -i2 -e8 -s1`
genome	`-g16384 -s64 -n16777216`
intruder	`-a10 -l128 -n262144 -s1`
kmeans high	`-i random-n65536-d32-c16.txt -m15 -n15 -t0.00001`
kmeans low	`-i random-n65536-d32-c16.txt -m40 -n40 -t0.00001`
labyrinth	`-i random-x512-y512-z7-n512.txt`
ssca2	`-s20 -i1.0 -u1.0 -l3 -p3`
vacation high	`-n4 -q60 -u90 -r1048576 -t4194304`
vacation low	`-n2 -q90 -u98 -r1048576 -t4194304`
yada	`-a15 -i ttimeu1000000.2`

Table 4. STAMP workloads

tions in Table 3. STAMP applications can be configured with different parameters to define different workloads. In our experiments, we use 10 workloads defined in the STAMP 0.9.10 distribution (Table 4). We choose STAMP because it offers a range of workloads and has been widely used to evaluate TM systems.

Micro-benchmarks. To evaluate low-level overheads of STMs, such as costs of synchronization and logging, with smaller-scale workloads, we use four micro-benchmarks from [18]. The benchmarks implement an integer set using different data structures: (1) hash table, (2) linked list, (3) red-black tree and (4) skip list. Every transaction executes a single lookup, insert or remove of a randomly chosen integer from the set. Initially, the data structures are filled with 2^{16} elements chosen among a range of 2^{17} values. During the experiments, 5% of the transactions are insert operations, 5% are remove operations, and 90% are search operations.

Privatization. None of the benchmarks we use requires privatization, which means that we measure the worst case: namely, supporting transparent privatization only incurs overheads, without the performance benefits of reading and writing privatized data outside of transactions as in [28].

2.3 Hardware

We used the following system configurations:

- Sun Microsystems UltraSPARC T2 with 32GB memory running Solaris 10. This machine has a single UltraSPARC T2 CPU that consists of 8 cores, each multiplexing 8 hardware threads, for a total of 64 supported hardware threads.
- AMD Opteron at 2.2GHz with 8GB memory running Linux kernel 2.6.22.19. This machine has four quad-core CPUs for a total of 16 supported hardware threads.

2.4 Experimental methodology

We repeated the execution of each experiment multiple times (at least five times) to reduce the variance in the results. We report averages from these runs.

2.5 Availability

SwissTM, STMBench7 and the code required to run SwissTM with other benchmarks are available at `http://lpd.epfl.ch/site/research/tmeval`.

The micro-benchmark suite we used is available at `http://lpd.epfl.ch/gramoli/estm`.

3. STM-ME Performance

Figure 2 depicts STM-ME (manual instrumentation with explicit privatization) speedup over sequential, non-instrumented code on SPARC. Values above x-axis mean that the STM version is faster, and below that it is slower than the sequential code. The figure shows that STM outperforms sequential code on all used benchmarks, by up to 29 times on `vacation low` benchmark. It scales up to 64 threads (the number of supported hardware threads) on `STMBench7 read`, `STMBench7 read/write`, `vacation`, `genome`, `kmeans low` and `ssca2` workloads. Although the performance stays the same or degrades from 32 to 64 threads on other benchmarks, STM has good overall performance on all used benchmarks. The data also shows that the less contention the workload exhibits, the more benefit we can expect from STM. For example, STM outperforms sequential code by more than 11 times on read dominated workload of STMBench7, and less than 2 times for write dominated workload of the same benchmark.

It is interesting to look at the number of concurrent threads that STM-ME requires to outperform sequential code. On SPARC, already with 2 concurrent threads, STM is faster than sequential code in 8 out of 17 considered workloads. With 4 threads this number raises to 14 workloads and with 8 STM-ME does not outperform sequential code only in `linked list` micro-benchmark. With 16 or more threads, STM is faster than sequential code in all considered workloads.

We would like to point out that while STM-ME outperforms sequential code in all the benchmarks, some of the achieved speedups are not very impressive (e.g. 1.4 times with 64 threads on `ssca2` benchmark). This just confirms

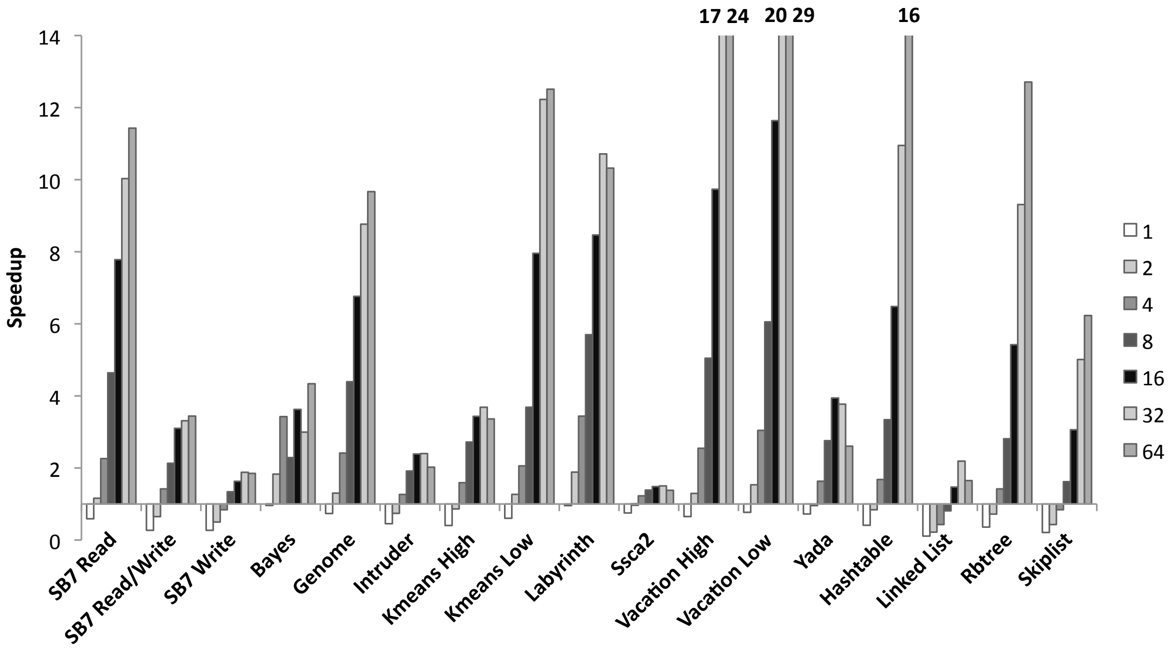

Figure 2. STM-ME performance with SPARC

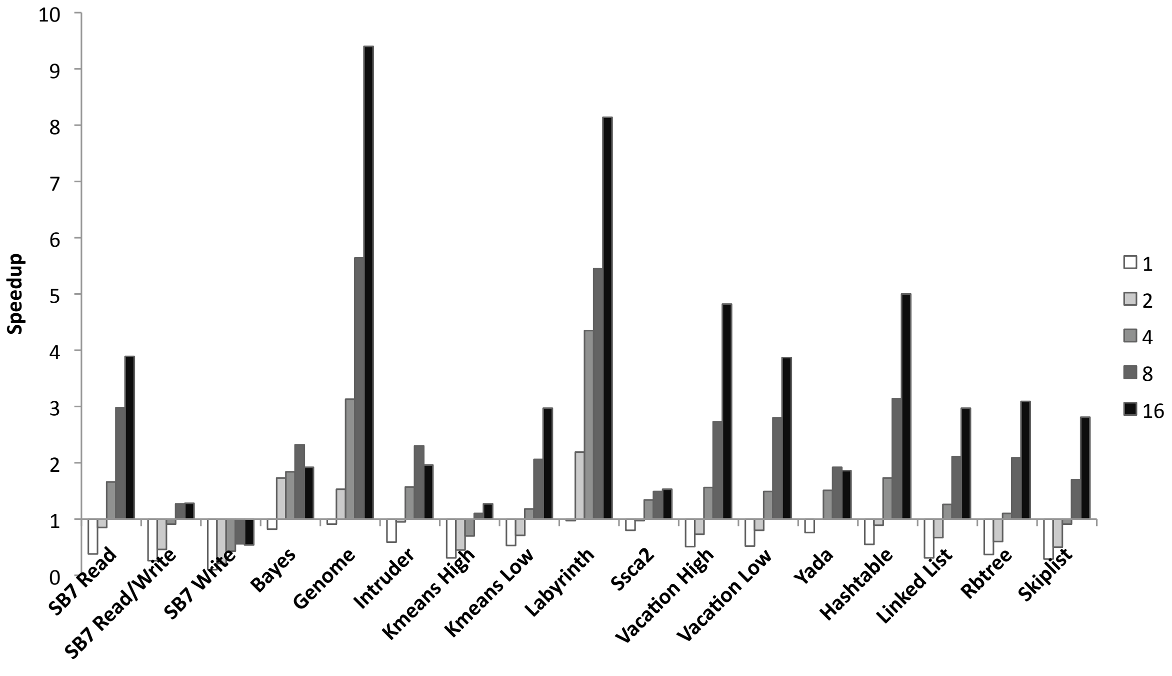

Figure 3. STM-ME performance with 16 core x86

that STM, while showing great promise for some types of concurrent workloads, is not the best solution for all of them.

On x86 (Figure 3), STM clearly outperforms sequential code in all workloads, except in the challenging write dominated STMBench7 workload. This workload consists of 90% operations that write some data, which results in very high contention. Performance gain when compared to sequential code is lower than on SPARC (up to 9 times on x86 compared to 29 times on SPARC). The reasons for this are two-fold: (1) all threads execute on the same chip with SPARC, so the costs of inter-thread communication is lower and (2) sequential performance of a single thread on SPARC is much lower.

On x86, STM-ME outperforms sequential code in 3 out of 17 workloads with 2 threads, and in 13 workloads with 4 threads. STM-ME is faster than sequential code in 16 workloads (all but `STMBench7 write`) with 8 and 16 threads.

To sum up, STM-ME scales well on both SPARC and x86 architectures. The absolute obtained performance is higher than sequential performance in all but one case. Furthermore, STM-ME outperforms sequential code in 13 out of 17 workloads already with 4 concurrent threads on both SPARC and x86. Thus, our results clearly show that STM-ME algorithms do scale and perform well in different settings.

Contradicting earlier results. The results of [8] indicate that STMs do not perform very well on three of the STAMP applications that we also have used: (1) `kmeans`, (2) `vacation`, and (3) `genome`. In our experiments STM has good performance on all three. In particular, STM-ME outperforms sequential code on all three benchmarks on both SPARC and x86 machines.

We believe reasons for such considerable difference between STM performance in our experiments and [8] are three-fold:

1. *Workload characteristics.* After getting the details of experimental settings of [8] from the authors, we noticed that their experiments did not use the default STAMP workloads. Configurations used in [8] significantly increase the contention in all three STAMP benchmarks, by reducing the size of the shared data in `kmeans` and `vacation` and reducing the fraction of read-only transactions in `genome`. Speculative approaches to concurrency, such as STM, do not perform well with very high contention workloads, which is confirmed by our experiments, in which STM has the lowest performance in very high contention benchmarks (e.g., `kmeans high` and `STMBench7 write`).

   To evaluate the impacts of workload characteristics on the performance, we ran STAMP workloads from [8] on a two quad-core CPU Xeon machine to evaluate the effects of different workload configurations on a machine that is similar (with exception of hyper-threading) to the one used in the other study (Figure 4). STM-ME performance is indeed lower than with the default STAMP parameters, but STM-ME outperforms sequential code in all three workloads.

   The performance we observed differs from the performance of [8] in several respects: (1) STM-ME outperforms sequential code in `vacation` with 8 threads, while it is slower than sequential code in the same benchmark in experiments of [8], (2) STM-ME scales well in `vacation` promising to keep improving performance as more cores are added, while it stops scaling at 4 threads in results from [8] and (3) STM-ME outperforms sequential code in `genome` by about 4.5 times with 8 threads (compared to about 2.5 times in [8]).

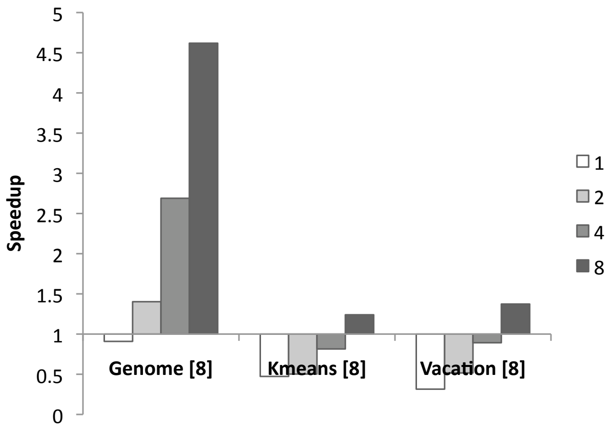

Figure 4. STM-ME on 8 core x86 with STAMP workloads from [8]

2. *Different hardware.* We used hardware configurations with the support for more hardware threads—64 and 16 hardware threads in our experiments compared to 8 in [8]. This lets STM perform better as there is more parallelism at the hardware level to be exploited.

   Also, our x86 machine does not use hyper-threading[2] while the one used in [8] uses a quad-core hyper-threaded CPU. Hardware thread multiplexing in hyper-threaded CPUs can degrade performance when compared to the machine supporting the same number of hardware threads without multiplexing.

   We ran the default STAMP workloads and the workloads from [8] on a machine with two single-core Xeon CPUs hyper-threading turned on (Figure 5) to evaluate impact of hyper-threading on performance. These results confirmed that hyper-threading indeed impacts the performance (and also scalability) of STM-ME. This (partially) explains performance difference with up to 8 threads when compared to [8].

3. *More efficient STM.* We believe that part of the performance difference comes from a more efficient STM implementation. The results of [14] suggest that SwissTM has better performance than TL2, which has comparable performance to IBM STM in [8].

To further evaluate STM performance, we also executed experiments with TL2 [11], McRT-STM [3] and TinySTM [35]. We were provided with the Bartok STM [1,

[2] Running multiple hardware threads on a single CPU core is also called hyper-threading on Intel's CPUs.

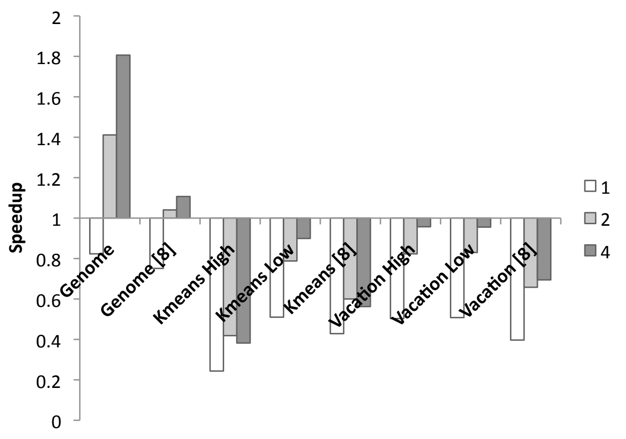

Figure 5. STM-ME on x86 with 2 CPUs and hyper-threading

23] performance results on a subset of STAMP by Tim Harris from Microsoft Research. For clarity of presentation we present the resulting graphs in the appendix. All of these experiments confirm our general conclusions about good STM performance on a wide range of workloads.

Further optimizations. In some of the workloads we used, performance degrades when too many concurrent threads are used. A solution to this problem would be to change the thread scheduler so it does not run more concurrent threads than is optimal.

Programming model. While using STM with explicit privatization and manual instrumentation yields very good performance, it also exposes a programming model that is not trivial to use, as it requires significant effort for manual instrumentation and care about explicitly dealing with privatization. As a result, STM-ME might be considered too tedious and error prone for use in most applications, and might be appropriate only for smaller applications or performance critical sections of the code. In the following, we discuss the cost of transparent privatization (Secion 4) and compiler instrumentation (Sections 5 and 6).

4. STM-MT Performance

The validation barriers scheme that we use for ensuring privatization safety requires frequent communication between all threads in the system and can degrade performance due to the time threads spend waiting for each other and the increased number of cache misses. A similar technique is already known to significantly impact performance of STM in certain cases [47], which our experiments confirm. It is worth repeating that none of our benchmarks uses privatization, so transparent privatization only impacts performance, without providing any benefit.

We show performance of STM-MT (manual instrumentation with transparent privatization) with SPARC in Figure 6.

	SPARC			x86		
Threads	Min	Max	Avg	Min	Max	Avg
1	0	0.06	0	0	0.45	0.08
2	0.02	0.47	0.16	0.03	0.58	0.29
4	0.03	0.59	0.26	0.06	0.64	0.4
8	0.03	0.66	0.32	0.08	0.69	0.48
16	0	0.75	0.35	0.17	0.85	0.51
32	0	0.77	0.34	-	-	-
64	0	0.8	0.35	-	-	-

Table 5. STM-MT overheads $(1 - \frac{perf_{STM-MT}}{perf_{STM-ME}})$

The figure conveys that transparent privatization impacts the performance of STM significantly, but that STM-MT still outperforms sequential code in all benchmarks.

The performance is, however, lower—STM-MT outperforms sequential code by up to 23 times compared to 29 times with STM-ME, and by 5.6 times on average compared to 9.1 times with STM-ME. The performance impact we observed confirms the results of [47].

Also, STM-MT requires more threads to outperform sequential code than STM-ME, as it performs better than sequential code in all workloads only with 64 threads. With 2 threads it is faster than sequential code on 5 workloads out of 17, with 4 threads on 11 workloads, with 8 threads on 13. With 16 and 32 threads STM-ME outperforms sequential code on all workloads but `STMBench7 read/write` and `STMBench7 write`.

Our experiments show that performance for some of the workloads is not impacted at all (e.g. `ssca2`), while the overheads can be as high as 80% (e.g. `vacation low`, `yada`). Also, in general, overheads increase with the number of concurrent threads, thus impacting both performance and scalability of STM. Table 5 summarizes the overheads of transparent privatization with SPARC.

We repeat the same experiments with x86 machine. The results are depicted in Figure 7. The data confirms that STM-MT has lower performance than STM-ME. Transparent privatization overheads reduce STM performance below performance of sequential code in 4 benchmarks—`STMBench7 read/write`, `STMBench7 write`, `kmeans high` and `hashtable`. On the first two benchmarks, the performance of STM-ME was close to sequential performance and transparent privatization reduces it enough to make it lower. The case of `hashtable` is more interesting as STM-ME performs quite well on it. We believe that transparent privatization cost is high on `hashtable` because of the cache contention for shared privatization metadata induced by small transactions.

Similarly to experiments on SPARC, STM-MT requires more threads to outperform sequential code on x86 than STM-ME. STM-MT outperforms sequential code in only 1 workload out of 17 with 2 threads and in 8 workloads with 4 threads. With 8 threads STM-MT is faster than sequential

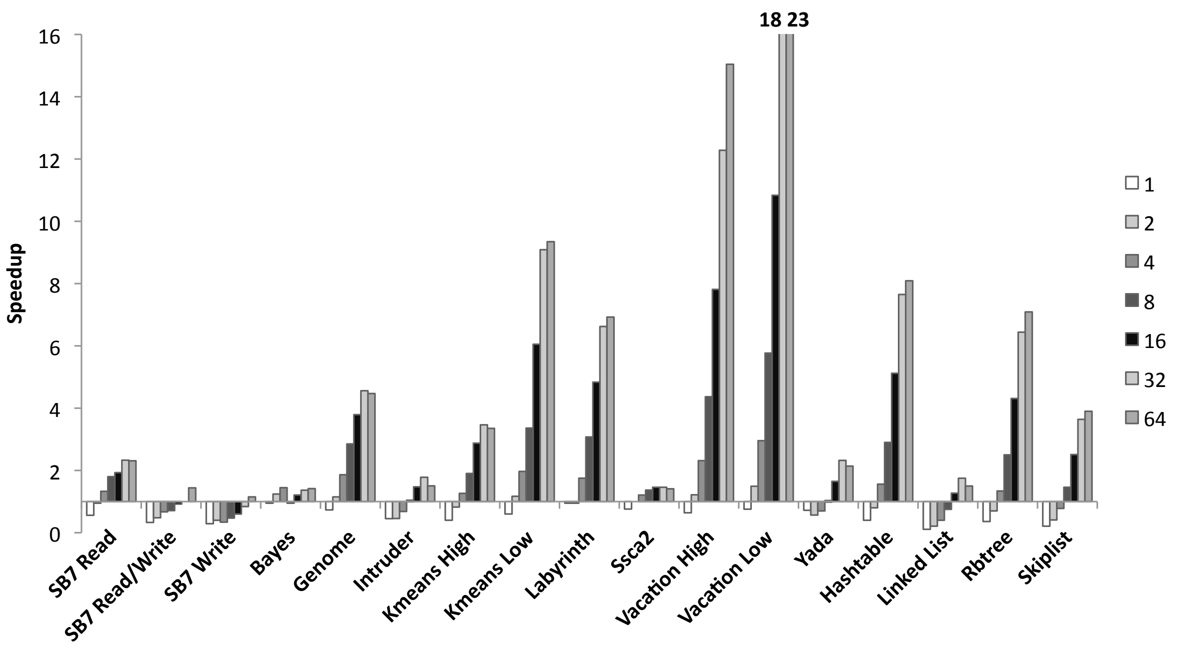

Figure 6. STM-MT performance with SPARC

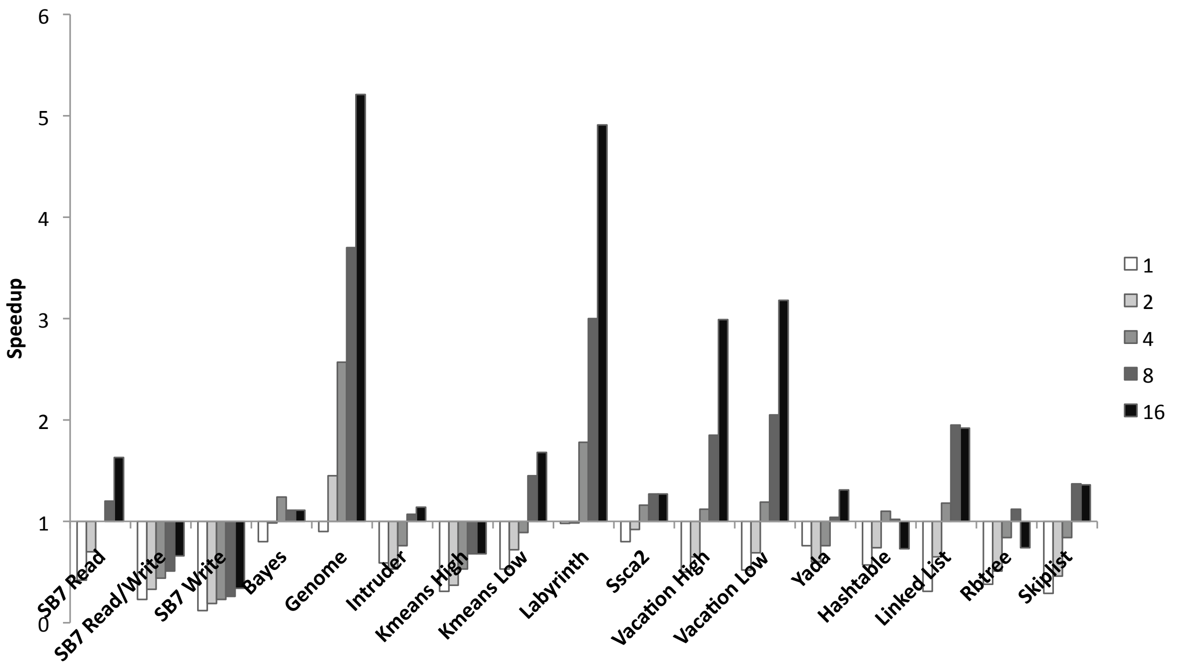

Figure 7. STM-MT performance with 16 core x86

code in 14 and and with 16 threads in 12 out of 17 workloads we used.

Our experiments show that privatization overheads can be as high as 80%. It also confirms that the transparent privatization overheads increase with the number of threads and that they have higher performance impact with workloads consisting of short transactions (like the micro-benchmarks we used). Overheads of transparent privatization are higher on our four-CPU x86 machine than on SPARC, mainly due to higher costs of inter-thread communication. The overheads of transparent privatization with x86 are summarized in Table 5.

To sum up, while the impact of transparent privatization can be significant, STM-MT can still scale and perform well on a wide range of applications. In particular, STM-MT outperforms sequential code on 13 out of 17 workloads with 8 concurrent threads on both SPARC and x86. Our conclusion is, furthermore, that reducing costs of cache coherence traffic by having more cores on a single chip reduces the costs of transparent privatization, resulting in better performance and scalability.

Further optimizations. It is known that the technique we used for ensuring privatization safety in STM does impact scalability and performance significantly in certain cases [47]. Two recent proposals [27, 31] aim to improve scalability of transparent privatization by employing partially visible reads. Partially visible reads allow writers to detect that some other transactions are reading memory words that the writers are updating and thus allow them to wait only for conflicting readers, if there are any, to ensure transparent privatization safety. By making readers only partially visible, the cost of reads is reduced, compared to fully visible reads, and the scalability of privatization support is improved. To implement partially visible readers, [31] uses timestamps, while [27] uses a variant of SNZI counters [17]. The main advantage of the approach used in [27] over the one in [31] is that [27] does not use centralized privatization meta-data which improves scalability significantly.

Programming model. We believe that while supporting transparent privatization relieves programmers of some issues, it might not be absolutely needed in an STM system. It seems that privatizing a piece of data is a conscious decision made by a programmer rather than an accident. This means that explicitly marking privatizing transactions would not require too much additional effort from the programmer. If this is indeed the case, it would allow for much less expensive explicit privatization support in STM implementations.

5. STM-CT Performance

Compiler instrumentation usually replaces more memory references by STM `load` and `store` calls than strictly necessary, resulting in the reduced performance of generated code (this is known as over-instrumentation [8, 15, 47]). Ideally, the compiler would only replace memory accesses with STM calls when they reference some shared data. However, the compiler does not have (1) information about all uses of variables in the whole program and (2) semantical information about variable use, which is typically available only to the programmer (e.g., which variables are private to some threads or which are read-only). This is why the compiler, conservatively, generates more STM calls than necessary. Unnecessary STM calls reduce performance because they are more expensive than the CPU instructions.

Threads	Min	Max	Avg
1	0	0.45	0.16
2	0.1	0.58	0.35
4	0.12	0.64	0.41
8	0.11	0.69	0.47
16	0.17	0.86	0.52

Table 6. STM-CT overheads with x86 $(1 - \frac{perf_{STM-CT}}{perf_{STM-ME}})$

We present STM-CT (compiler instrumentation with transparent privatization) speedup over sequential code in Figure 8.[3] Despite high overheads of transparent privatization and compiler over-instrumentation, STM-CT outperforms sequential code in all workloads but `intruder`, `kmeans high` and `rbtree`.

However, it requires higher thread counts to outperform sequential code than STM-MT for the same workloads. STM-CT is faster than sequential code only on 1 out of 14 workloads with 2 threads (`genome`). With 4 threads STM-CT outperforms sequential code on 5 workloads, and with 8 threads on 8 workloads. STM-CT is faster than sequential code in all but 4 workloads with 16 threads.

As expected, the overheads of STM-CT are higher than the overheads of STM-MT. The overheads can be as high as 80% (on `hashtable` and `rbtree`), but also as low as 20% (on `ssca2` and `linked list`). Table 6 summarizes the overheads of STM-CT when compared to STM-ME.

Programming model. STM-CT exposes a simple programming model by using both compiler instrumentation and transparent privatization. As a drawback, its performance is significantly worse than the performance of STM-ME or STM-MT. If our intuition with respect to transparent privatization being unnecessary is correct, then it is worth investigating a slightly weaker programming model—STM variant that uses compiler instrumentation and explicit privatization.

6. STM-CE Performance

We present STM-CE (compiler instrumentation with explicit privatization) speedup over sequential code in Figure 9.[4] The figure shows that STM-CE outperforms sequential code in all benchmarks but `kmeans high`. However, it scales well on `kmeans high` and promises to outperform sequential code with additional hardware threads.

STM-CE outperforms sequential code with fewer threads than both STM-MT and STM-CT. It performs better than sequential code in 4 out of 14 workloads already with 2

[3] The data we present here is only for x86 and it does not include STMBench7 workloads, due to the limitations of the STM compiler we used.

[4] The data we present here is only for x86 and it does not include STMBench7 workloads, due to the limitations of the STM compiler we used.

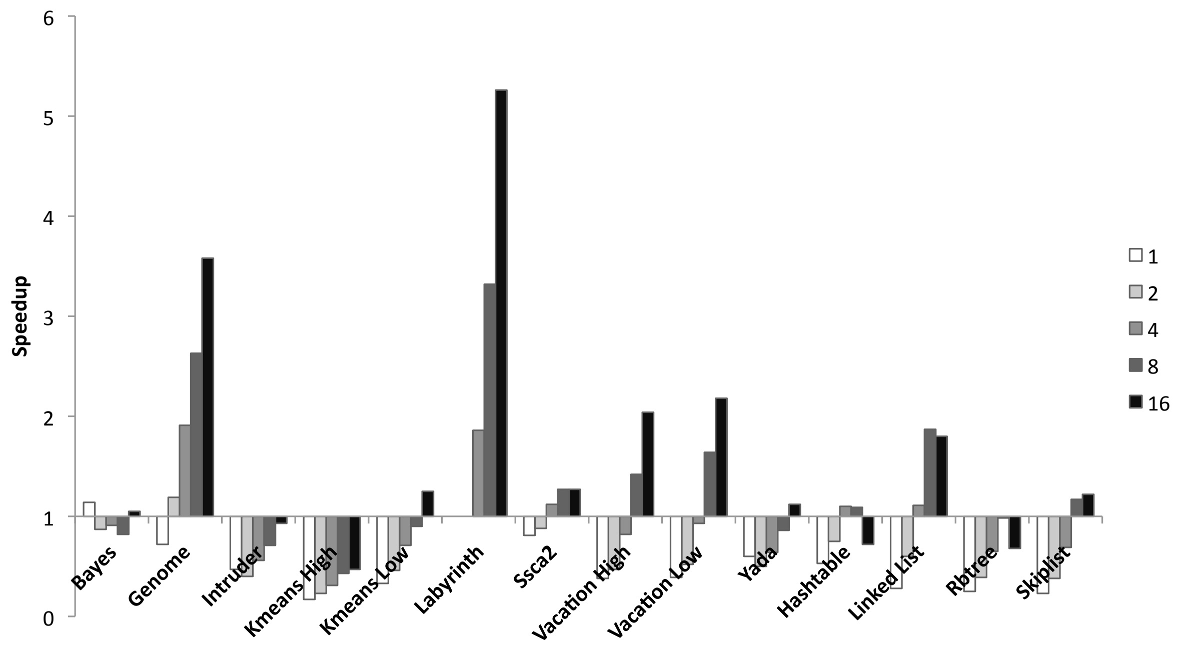

Figure 8. STM-CT performance with 16 core x86

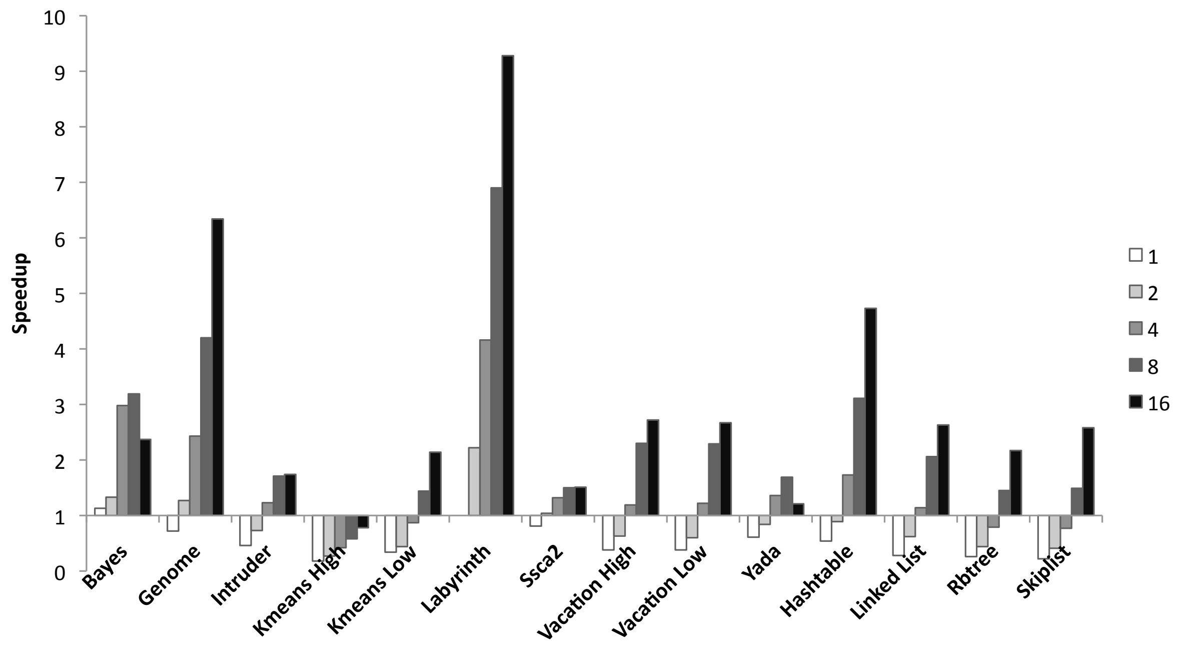

Figure 9. STM-CE performance with 16 core x86

threads. [5] With 4 threads STM-CE outperforms sequential code in 10 workloads and with 8 and 16 threads on 13 workloads (all but `kmeans high`).

[5] This is better even than STM-ME, but only due to slight variations in the collected data.

The overheads of compiler over-instrumentation remain around 20% for all workloads but `kmeans` where they are about 40%. Also, on some workloads (`labyrinth`, `ssca2` and `hashtable`) compiler instrumentation does not introduce significant overheads and the performance of STM-ME

Threads	Min	Max	Avg
1	0	0.42	0.16
2	0	0.4	0.17
4	0	0.4	0.11
8	0	0.47	0.11
16	0	0.44	0.17

Table 7. STM-CE overheads with x86 $(1 - \frac{perf_{STM-CE}}{perf_{STM-ME}})$

and STM-CE is almost the same. It is interesting to note that the overheads of compiler instrumentation remain approximately the same for all thread counts, conveying that compiler instrumentation does not impact STM scalability. Table 7 summarizes overheads introduced by the compiler instrumentation.

To summarize, the additional overheads introduced by compiler instrumentation remain acceptable as performance is worse than sequential in only one of the tested workloads. Moreover, with only 4 threads STM-CE outperforms sequential code on 10 out of 14 workloads we experimented with.

Further optimizations. A great deal of recent work focuses on leveraging compiler optimization to improve performance of compiler instrumented STM applications. We briefly describe some of this work here.

In [33], optimizations that replace full STM `load` and `store` calls with specialized, faster versions of the same calls are described. For example, some STMs can perform fast reads of memory locations that were previously accessed for writing inside the same transaction. While the compiler we used supports these optimizations, we did not implement the lower cost STM barriers in SwissTM yet. Compiler data structure analysis (DSA) is used in [37] to optimize the code generated by Tanger STM compiler. The optimization proposed identifies special types of data partitions at compile time, which allows it to use better suited forms of concurrency control than the generic STM algorithm. For example, read-only partitions are identified, and no concurrency control is required when accessing read-only data.

Several optimizations have been proposed in context of Java [3] that eliminate transactional accesses to immutable data and data allocated inside current transaction. Bartok-STM [23] uses flow-sensitive inter-procedural compiler analysis and runtime log filtering to identify objects allocated in the current transaction and eliminate transactional accesses to them. In [16] dataflow analysis is used to eliminate some unnecessary transactional accesses. These approaches are more effective than what our compiler does because they are implemented in the context of managed languages.

Programming model. We believe (as well as others [9]) that the STM compiler is crucial for an STM system to be easy-to-use. Furthermore, if the programmers could indeed use explicit privatization easily enough, STM-CE is the programming model that is both easy to use and has a reasonable performance in a wide range of scenarios. This means that STM-CE could be used for real-world systems in the near future.

7. Conclusion

Our experimental results show that an STM can outperform sequential code across a wide range of workloads and on two different multi-core architectures. Our conclusions significantly differ from a recent study [8], which expressed strong doubts about STM performance. Whereas we do not argue that STMs are a silver bullet for general purpose concurrent programming, our results suggest that STM is already now usable for some types of applications. Current research on improving STM performance would only improve matters, and could only widen the range of applications in the near future. Also, research in Hybrid TM that aims at combining hardware and software implementations of transactions will benefit from achievements in the STM area.

Even though we believe STM is a mature research topic there is still room for improvements. For example, static segregation of memory locations depending on whether they are shared or not can minimize compiler instrumentation overhead, partially visible reads can improve privatization performance and reduction of accesses to shared data can enhance scalability.

Acknowledgments

We are grateful to Tim Harris for running the Bartok-STM experiments on the Intel Xeon machine, to Calin Cascaval for providing us with the experimental settings of [8] and to Yang Ni for running experiments with McRT-STM that in part confirmed our speculations about the impact of hyper-threading. We also would like to thank Hillel Avni, Derin Harmanci, Michał Kapałka, Patrick Marlier, Maged Michael and Mark Moir for fruitful discussions and comments. This research is funded by the FP7 Integrated Approach to Transactional Memory on Multi-and Many-core Computers (VELOX) project (ICT-216852) supported by the European Commission.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09*.

[2] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. E. Saha. Unlocking concurrency: Multicore programming with transactional memory. *ACM Queue*, 4(10), Dec 2006.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06*.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05*.

[5] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP '08*.

[6] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ISCA '08*.

[7] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*.

[8] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[9] L. Dalessandro, V. J. Marathe, M. F. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Transact '07*.

[10] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09*.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06*.

[12] D. Dice and N. Shavit. Tlrw: Return of the read-write lock. In *Transact '09*.

[13] D. Dice and N. Shavit. What really makes transactions faster? In *Transact '06*.

[14] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09*.

[15] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing transactions for captured memory. In *SPAA '09*.

[16] G. Eddon and M. Herlihy. Language support and compiler optimizations for STM and transactional boosting. In *ICDCIT '07*.

[17] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: scalable nonzero indicators. In *PODC '07*.

[18] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC'09*.

[19] D. Grossman. The transactional memory / garbage collection analogy. *SIGPLAN Not.*, 2007.

[20] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *PODC '05*.

[21] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *EuroSys '07*.

[22] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03*.

[23] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06*.

[24] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03*.

[25] M. Herlihy, M. Moir, and V. Luchangco. A flexible framework for implementing software transactional memory. In *OOPSLA '06*.

[26] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[27] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Transact '09*.

[28] L. D. M. L. Scott, M. F. Spear and V. J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07*.

[29] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *DISC '05*.

[30] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *Transact '06*.

[31] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP '08*.

[32] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *HPCA '06*.

[33] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA '08*.

[34] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05*.

[35] T. Riegel, P. Felber, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08*.

[36] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06*.

[37] T. Riegel, C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. In *SPAA '08*.

[38] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*.

[39] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP '07*.

[40] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*.

[41] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*.

[42] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing, Special Issue*, 10:99–116, 1997.

[43] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42(6):78–88, 2007.

[44] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC '07*.

[45] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC '06*.

[46] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update. In *SPAA '07*.

[47] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08*.

A. Performance of other STMs

In this appendix, we provide scalability data for several other STMs that further support our conclusions.

A.1 TinySTM

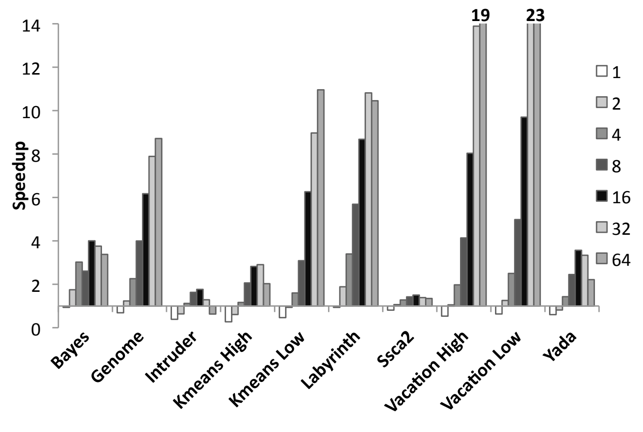

Figure 10. TinySTM performance with SPARC

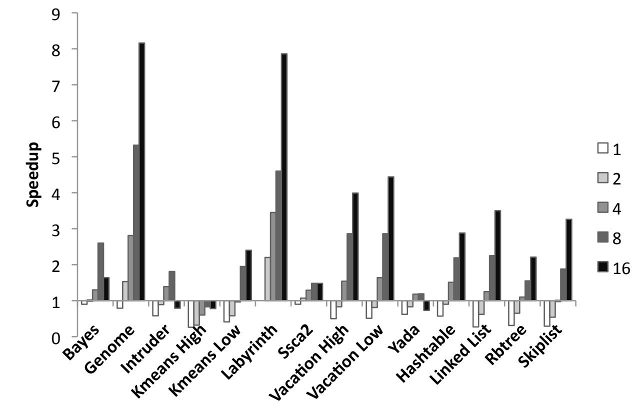

Figure 11. TinySTM performance with 16 core x86

Sequential speedup of TinySTM both SPARC and x86 is depicted in Figures 10 and 11. We used manually instrumented benchmarks with version of TinySTM that supports explicit privatization. TinySTM scales well and outperforms sequential code in all STAMP benchmarks on SPARC. On x86 machine, it outperforms sequential code in all benchmarks but `kmeans high`.

A.2 TL2

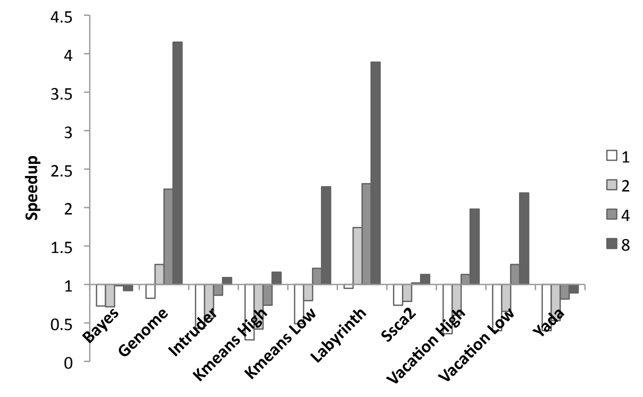

Figure 12. TL2 performance with 8 core x86

Sequential speedup of x86 TL2 version on STAMP benchmarks is depicted in Figure 12. We used manually instrumented STAMP benchmarks with version of TL2 that supports explicit privatization. The data supports our case as all applications (except `bayes` which exhibits highly varying execution times) scale well. Also, the only application for which TL2 does not outperform sequential code, although it comes close, with 8 threads is `yada`.

A.3 McRT STM

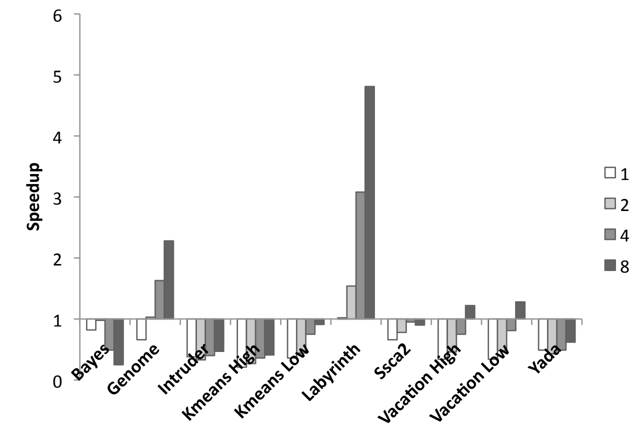

Figure 13. McRT STM C/C++ performance on STAMP with 8 core x86

The data showing sequential speedup of Intel's McRT C/C++ STM for STAMP applications is shown in Figure 13. We use a STM-CT version of the STM as this is the only version we had access to. The figure supports our previous experiments with STM-CT—although the performance suffers significantly from privatization and over-instrumentation overheads STM-CT still outperforms sequential code in several workloads. Still McRT STM with eight threads is able to outperform sequential code in several benchmarks.

A.4 Bartok STM

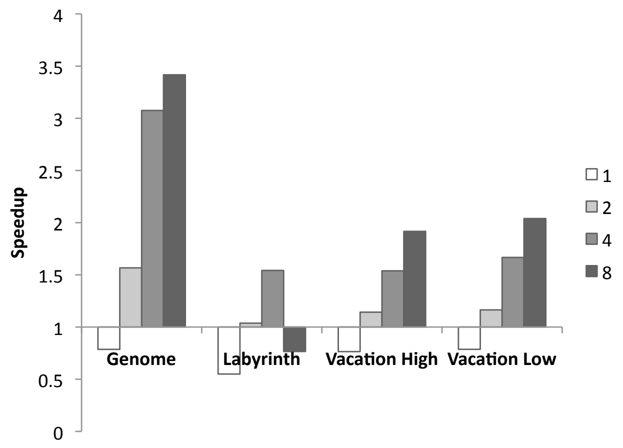

Figure 14. Bartok-STM performance with 8 core x86

Figure 14 depicts scalability of Bartok STM [1, 23] on a subset of STAMP benchmarks when run on two quad-core Intel Xeon 5300-series CPUs.[6] STAMP applications were ported to C# in the way described in [1]. The figure shows that both the scalability and the performance of Bartok STM are quite good. Except for `labyrinth` benchmark, performance scales to the number of available hardware threads (performance drop with `labyrinth` might be due to stop-the-world GC the runtime uses). Also the STM outperforms sequential code already with two threads in all benchmarks.

[6] The results were kindly provided by Tim Harris from Microsoft Research.