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 enables to leverage multicores and outperform sequential code. A recent CACM paper [3] questioned this ability and suggested the confinement of STM to a research toy.

We revisit these 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 concurrent 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 [4]. However, HTMs are so far restrictive as they can only handle transactions of limited size, or require some system events or CPU instructions to be executed outside transactions [4]. While there have been attempts to address these issues (e.g., [19]), 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) [15, 23] circumvents the limitations of HTM by implementing TM functionality fully in software. Furthermore, several STM implementations are already freely available and appealing for concurrent programming (e.g., [1, 5, 7, 11, 14, 20]). 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 book-keep data about every access. STMs check for conflicts, log the access, and in case of a write, log the current (or old) value of the data. Some of these operations use expensive synchronization instructions and access shared meta-data, which further increases their costs.
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 can be performed by an STM *compiler*. With a compiler, programmers only need to specify which sequences of statements have to be atomic, by enclosing them in transactional blocks. The compiler generates code that invokes appropriate STM read/write calls. While an STM compiler significantly reduces programming complexity, it can degrade performance of resulting programs (when compared to manual instrumentation) due to over-instrumentation [3, 8, 25]: basically, the compiler instruments the code conservatively with unnecessary calls to STM functions, as it cannot precisely determine which instructions indeed access shared data.
3. **Transparent privatization.** Making certain shared data private to a certain thread is known as *privatization*. Pri-

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

Table 1. STM support

vatization is typically used to allow non-transactional accesses to some data, either to support legacy code or to improve performance by avoiding costs of STM calls when accessing private data. Unless certain precautions are taken, privatization can result in race conditions [24]. Two approaches to prevent these have been considered: (1) a programmer *explicitly* marks transactions that privatize data, or (2) the STM *transparently* ensures that all transactions safely privatize data. Explicit privatization places additional burden on the programmer, while transparent privatization incurs runtime overheads [25], especially in cases when no data is actually being privatized.

Several research papers have discussed the scalability of STM with the increasing number of threads e.g., [1–3, 5, 7, 13, 16, 18, 20]. Very few however compared STM to sequential code and actually addressed the question of whether STM can be a viable option for speeding up the execution of applications.

Two notable exceptions are [2] and [3]. In [2], STM is shown, on a hardware simulator, to outperform sequential code in most STAMP benchmarks. Recently, [3] exhibited a series of experiments on a real hardware where STM performed worse than sequential code, and implied by their title that STM is only a "research toy". A closer look at the experiments revealed however that they considered a subset of the STAMP benchmark suite, configured in a specific manner and using only up to 8 threads.

We went a step further and compared STM performance to sequential code using (1) a larger and more diverse set of benchmarks and (2) real hardware that supports higher levels of concurrency. More specifically, we experimented with a state-of-the-art STM implementation, SwissTM [7], running three different STMBench7 [12] workloads, all ten workloads of the STAMP (0.9.10) [2] benchmark suite, as well as four micro-benchmarks, all encompassing both large and small scale workloads. We considered 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. Finally, we also considered all combinations of privatization and compiler support for STM (summarized in Table 1). Altogether, this constitutes the most to date exhaustive performance comparison of STM to sequential code.

Our experiments (summarized in Table 2) show that STM indeed outperforms sequential code in most configurations and benchmarks, offering already now a viable paradigm for concurrent programming. Maybe even more importantly, STM performs well with a small number of threads on many benchmarks. For example, STM-ME outperforms sequential code already with 4 threads on 14 and 13 out of 17 workloads on our SPARC and x86 machines respectively. Basically, we support the initial hopes about the good performance of STM, and we motivate further research in the field. We contradict the results of [3], and we believe we do so for three reasons: (1) STAMP workloads used in [3] present higher contention than the default STAMP workloads, (2) we use hardware that supports higher numbers of threads and, in case of x86, that does not use hyper-threading, and (3) we used a state-of-the-art STM implementation more efficient than those used in [3].

Clearly, and despite rather good STM performance in our experiments, there is still room for improvements. We highlight promising directions throughout the paper. Also, while there are several programming issues with the use of STM [3], (e.g., ensuring weak or strong atomicity, semantics of privatization, support for legacy binary code, etc), alternative concurrency programming approaches, like fine-grained locking or lock-free techniques, are not easier to use than STM. Such comparisons have been discussed in [11, 13, 15, 23] and are outside of the scope of this paper.

In the following, we first detail our evaluation settings, then present and discuss the experimental results for all four STM variants, and finally conclude.

2. Evaluation settings

In this section, we overview the STM library used for our experimental evaluation, SwissTM [7], as well as the benchmarks and hardware settings. Note that our experiments with other state-of-the-art STMs [5, 14, 18, 20] confirm our results [6]. SwissTM and the used benchmarks are available at `http://lpd.epfl.ch/site/research/tmeval`.

2.1 SwissTM

Synchronization algorithm. SwissTM [7] is a word-based STM that uses invisible (optimistic) reads. It relies on a time-based scheme to speed up read-set validation, similarly to [5, 21]. SwissTM detects read/write conflicts lazily and write/write conflicts eagerly. The two-phase contention manager uses different algorithms for short and long transactions. This design was carefully chosen to provide good performance across a wide range of workloads [7].

Privatization. We implemented privatization support in SwissTM using a simple validation barriers scheme described in [24]. To ensure safe privatization, each thread, after committing transaction T, waits for all other concurrent transactions to commit, abort or validate before executing application code after T.

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

Table 2. Summary of STM speedup over sequential code

Compiler instrumentation. We used Intel's C/C++ STM compiler [1, 18] for generating compiler instrumented benchmarks.[1]

2.2 Benchmarks

STMBench7. STMBench7 [12] is a synthetic STM benchmark that models realistic large-scale CAD/CAM/CASE workloads. 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 are its large data structure and long transactions in comparison to other typical STM benchmarks. In this sense, STMBench7 is very challenging for STM implementations.

STAMP. STAMP [2] offers a range of workloads and has been widely used to evaluate TM systems. It consists of 8 different applications representative of real-world workloads. STAMP applications can be configured with different parameters defining different workloads. In our experiments, we use 10 workloads from the STAMP 0.9.10 distribution. These include low and high contention workloads for `kmeans` and `vacation` applications and one workload for all other applications. The exact workload settings we used are specified in the companion technical report [6].

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 that implement an integer set using different data structures. Every transaction executes a single lookup, insert or remove of a randomly chosen integer from value range. 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.

It is important to note that the described benchmarks are TM benchmarks, and, thus, most of them use transactions extensively (with the exception of `labyrinth` and to a lesser extent `genome` and `yada`). Applications that would use transactions to simplify synchronization, but in which only a small fraction of execution time would be spent in transactions, would benefit from STM more than the benchmarks used. In a sense, the benchmarks we use represent a worst-case scenario for STM usage.

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

We report on our experiments in the following sections. For each experiment, we compute averages from at least five runs.

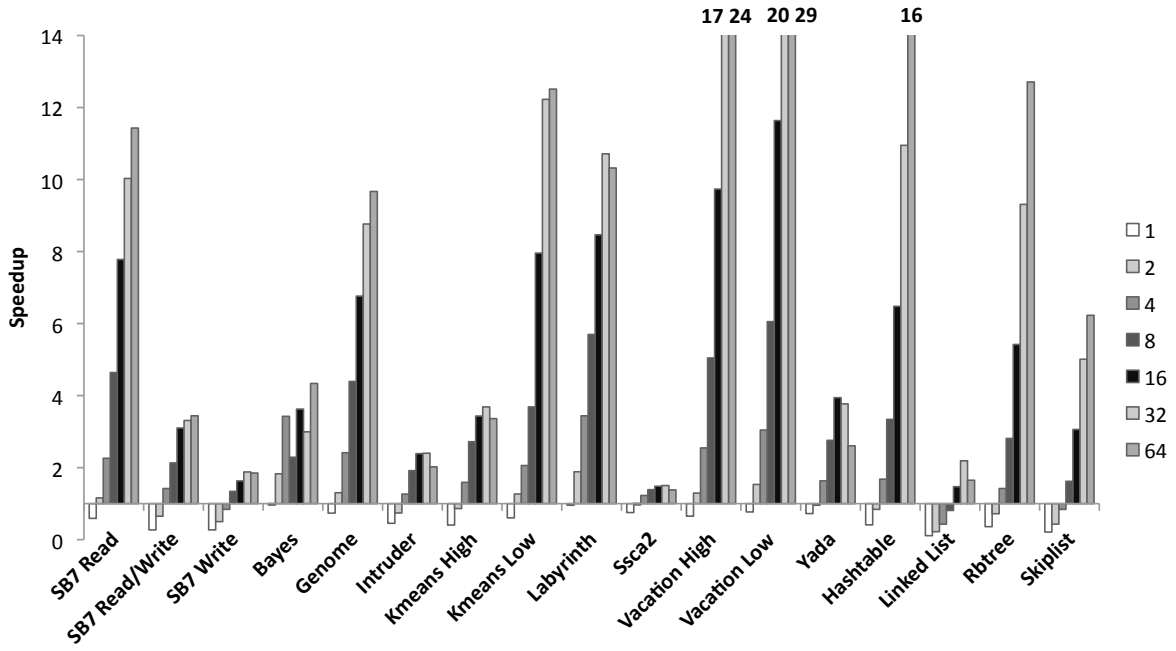
3. STM-ME Performance

Figure 1(a) depicts STM-ME (manual instrumentation with explicit privatization) speedup over sequential, non-instrumented code on SPARC. The figure shows that STM-ME has good performance already with a small number of threads, outperforming sequential code on 14 out of 17 workloads with 4 threads. The figure further shows that STM outperforms sequential code on all used benchmarks, by up to 29 times on `vacation low` benchmark. The experiment shows that the less contention the workload exhibits, the more benefit we can expect from STM, e.g., 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.

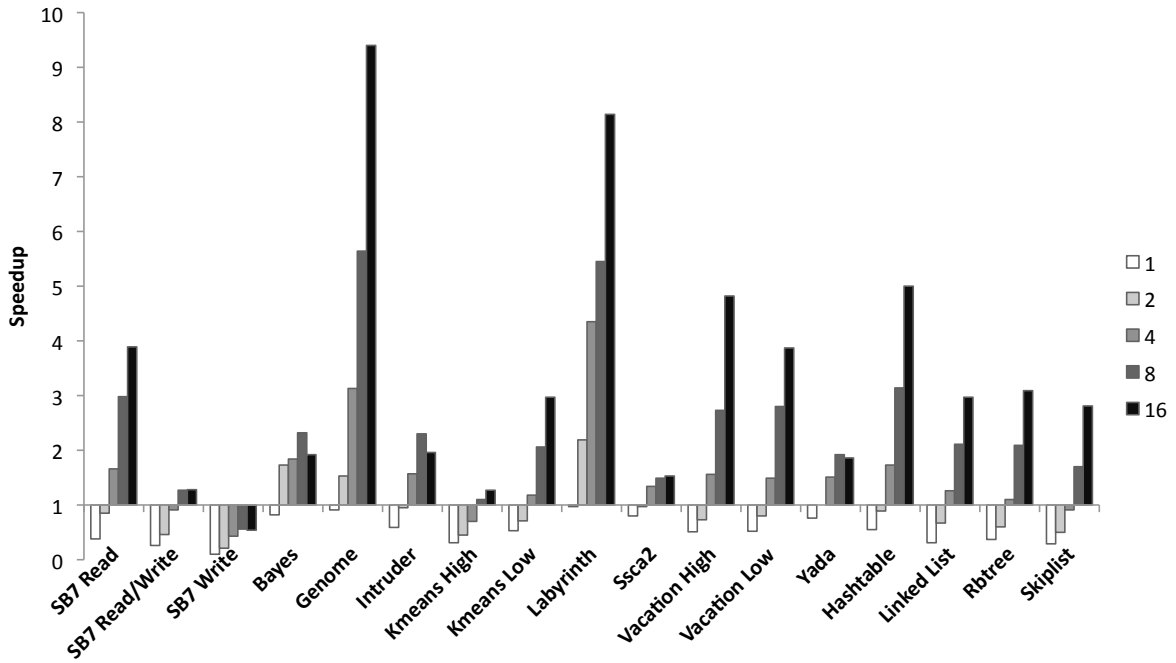
On x86 (Figure 1(b)), STM-ME outperforms sequential code on 13 workloads already with 4 threads. Also, STM clearly outperforms sequential code in all workloads, except in the challenging, high contention `STMBench7 write` workload. The performance gain, when compared to sequential code, is lower than 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.

To summarize, STM-ME has good performance on both SPARC and x86 architectures, clearly showing that STM-ME algorithms can scale and perform well in different settings. It is however important 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 that STM, while showing great promise for some types of concurrent workloads, is not the best solution for all of them.

Contradicting earlier results. The results of [3] indicated that STMs do not perform 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. The reasons for such considerable difference are, we believe, three-fold:



(a) SPARC

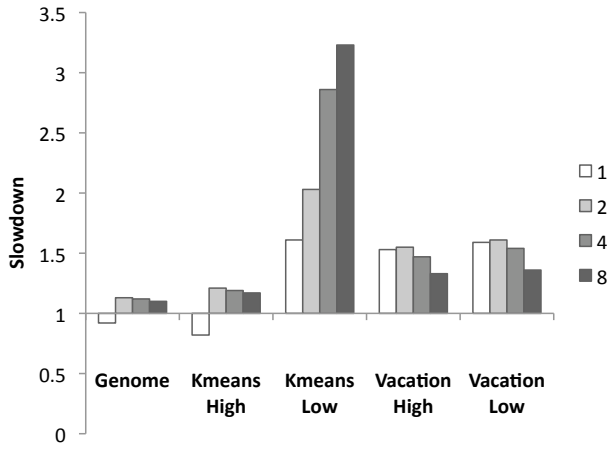


(b) x86

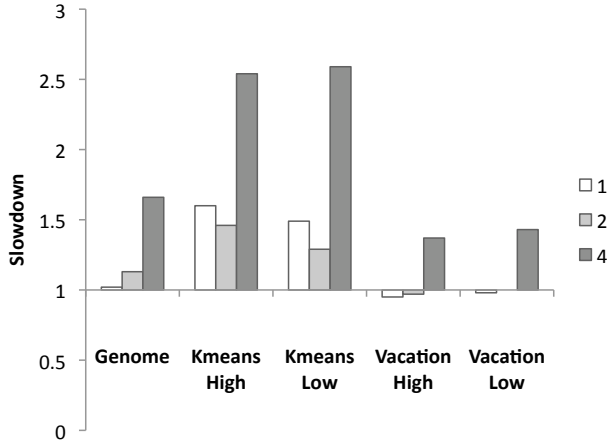
Figure 1. STM-ME performance

1. *Workload characteristics.* A closer look at the experimental settings of [3] reveal that their workloads had higher contention than the default STAMP workloads. STM usually has the lowest performance in highly contended workloads, consistent with our previous experiments (Figure 1).

   To evaluate the impacts of workload characteristics, we ran both default STAMP workloads and STAMP work-



(a) workload impact



(b) hyper-threading impact

Figure 2. Impact of different experimental settings of [3] on STM-ME performance

loads from [3] on a 2 quad-core CPU Xeon machine (which is more similar to the machine used in [3] than the x86 machine we used in other experiments). Slowdown of workloads from [3] compared to default STAMP workloads (we used both low and high contention workloads for `kmeans` and `vacation`) is depicted in Figure 2(a). Workload settings from [3] indeed degrade performance of STM-ME. The performance impact is significant in `kmeans` (around 20% for high- and up to 200% for low-contention workload) and in `vacation` (30% to 50%in both workloads). The performance is least impacted in `genome` (around 10%).

2. *Different hardware.* We used hardware configurations with support for more hardware threads—64 and 16 hardware threads in our experiments compared to 8 in [3]. 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 while the one used in [3] does. Hardware thread multiplexing in hyper-threaded CPUs can hamper performance. To evaluate this impact, we ran the default STAMP workloads on a machine with 2 single-core hyper-threaded Xeon CPUs. Figure 2(b) depicts the slowdown on the hyper-threaded machine compared to the similar machine without hyper-threading. The figure shows that hyper-threading impacts performance significantly, especially with higher thread counts. Slowdown in `genome` with 4 threads is around 65% and in two `vacation` workloads around 40%. The performance difference in `kmeans` workloads is significant even with a single thread, which is due to differences in CPUs that are not related to hyper-threading. Still, even with `kmeans`, slowdown with 4 threads is much higher than with 1 and 2 threads.

3. *More efficient STM.* We also believe that part of the performance difference comes from a more efficient STM implementation. The results of [7] suggest that SwissTM has better performance than TL2, which performs comparably to the IBM STM in [3].

We also experimented with TL2 [5], McRT-STM [1] and TinySTM [20]. We were provided with the Bartok STM [14] performance results on a subset of STAMP by Tim Harris from Microsoft Research. All of these experiments confirm our general conclusions about good STM performance on a wide range of workloads [6].

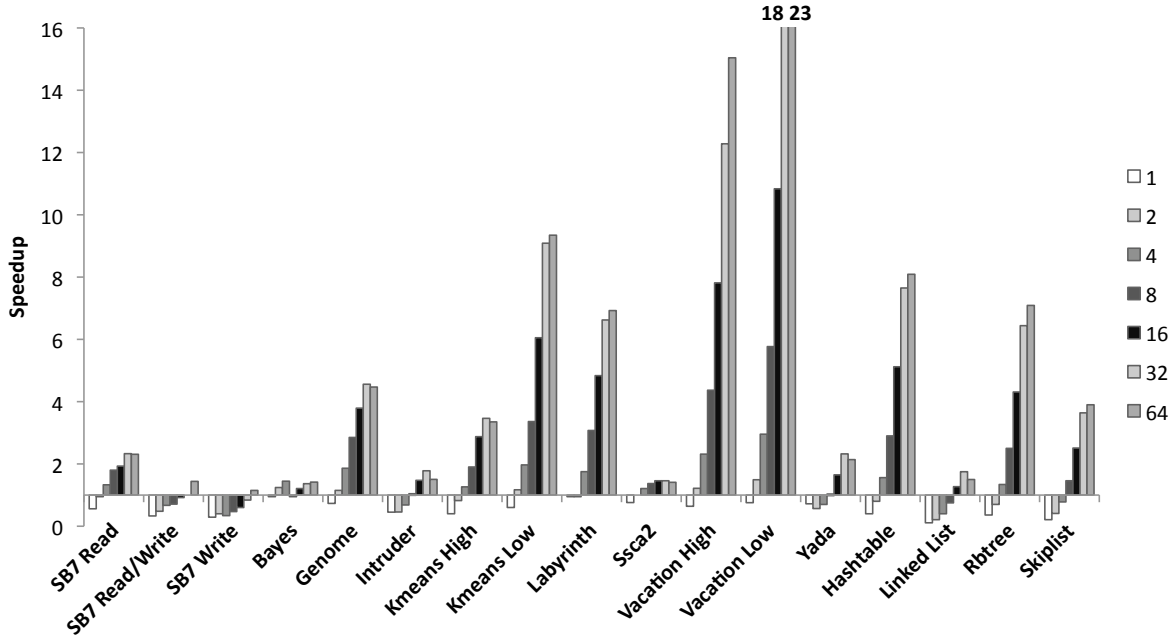
Further optimizations. In some of the workloads we used, performance degrades when too many concurrent threads are used. One possible way out would be to modify the thread scheduler so that it avoids running more concurrent threads than is optimal for a given workload, based on the information provided by the STM runtime.

4. STM-MT Performance

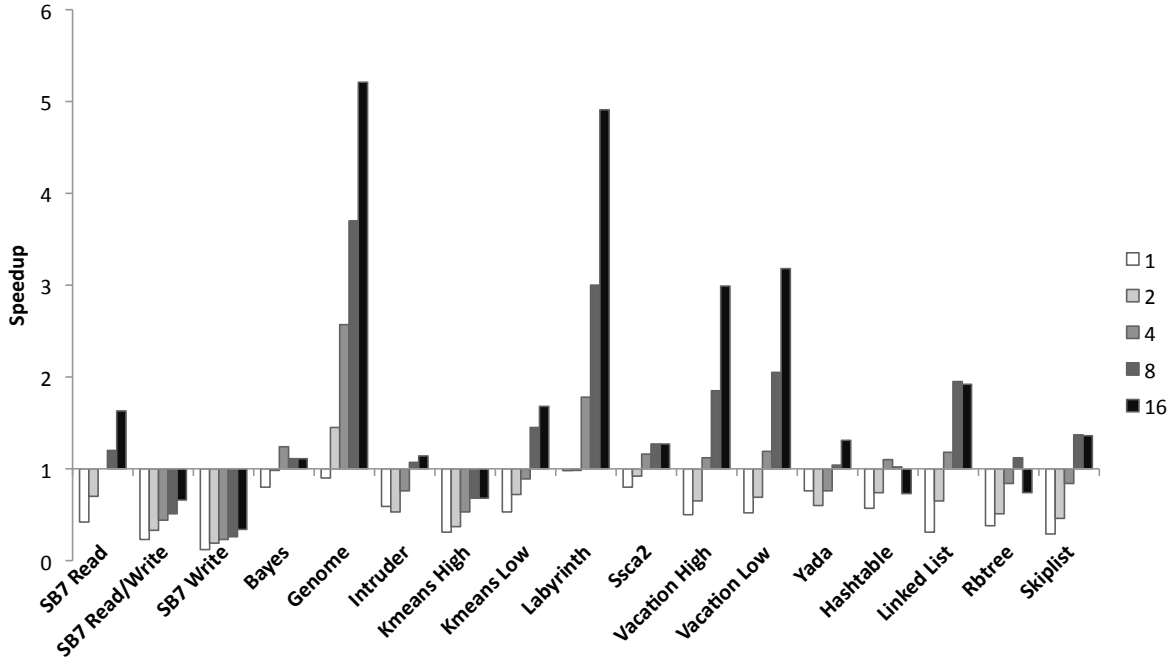
Validation barriers that we use for ensuring privatization safety require frequent communication between all threads in the system and can degrade performance due to the time threads spend waiting for each other and the increased number of cache misses. A similar technique is already known to significantly impact performance of STM in certain cases [25], which our experiments confirm.

It is important to highlight here the very fact that none of the benchmarks we use requires privatization. We thus measured the worst case: supporting transparent privatization only incurs overheads, without the performance benefits of reading and writing privatized data outside of transactions. Also, the measured performance costs are specific to our choice of privatization technique and implementation, and proposals for reducing privatization costs exist [16, 17].

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



(a) SPARC



(b) x86

Figure 3. STM-MT performance

ure 3(a). The figure conveys that transparent privatization impacts the performance of STM significantly, but that STM-MT still performs well, managing to outperform sequential code on 11 out of 17 workloads with 4 threads and on 13 workloads with 8 threads. Also, 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

	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 3. Transparent privatization cost ($1 - \frac{speedup_{STM\text{-}MT}}{speedup_{STM\text{-}ME}}$)

by 5.6 times on average compared to 9.1 times with STM-ME.

Our experiments show that performance for some of the workloads is not impacted at all (e.g., `ssca2`), while the privatization costs can be as high as 80% (e.g., `vacation low`, `yada`). Also, in general, costs increase with the number of concurrent threads, thus impacting both performance and scalability of STM. Table 3 summarizes the costs of transparent privatization with SPARC.

We repeated the same experiments with the x86 machine (Figure 3(b)). The data confirms that STM-MT has lower performance than STM-ME. It outperforms sequential code on 8 out of 17 workloads with 4 threads and on 14 workloads with 8 threads. Overall, transparent privatization overheads reduce STM performance below performance of sequential code in 3 benchmarks—`STMBench7 read/write`, `STMBench7 write` and `kmeans high`. It is interesting to note that the performance is impacted the most with micro-benchmarks. We believe that this is due to cache contention for shared privatization meta-data induced by small transactions.

Our experiments show that privatization costs can be as high as 80%. It also confirms that the transparent privatization costs increase with the number of threads. Costs of transparent privatization are higher on our four-CPU x86 machine than on SPARC, mainly due to higher costs of inter-thread communication. The costs of transparent privatization with x86 are shown in Table 3.

To summarize, while the impact of transparent privatization can be significant, STM-MT can still scale and perform well on a wide range of applications. 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. Two recent proposals [16, 17] aim to improve scalability of transparent privatization by employing partially visible reads. 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, [17] uses timestamps, while [16] uses a variant of SNZI counters [10]. In addition, [16] avoids using centralized privatization meta-data to improve scalability.

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 4. Compiler instrumentation cost with x86 ($1 - \frac{speedup_{STM\text{-}CE}}{speedup_{STM\text{-}ME}}$)

5. STM-CE Performance

Compiler instrumentation often 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 [3, 8, 25]). 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) semantic 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 that they replace.

We present STM-CE (compiler instrumentation with explicit privatization) speedup over sequential code in Figure 4.[2] The figure shows that STM-CE has good performance, as it outperforms sequential code on 10 out of 14 workloads with 4 threads and on 13 workloads with 8 threads. Overall, it outperforms sequential code in all benchmarks but `kmeans high`. However, it scales well on `kmeans high` promising to outperform sequential code with additional hardware threads.

The costs of compiler 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 costs and the performance of STM-ME and STM-CE is almost the same. It is interesting to note that, in our experiments, the costs of compiler instrumentation remain approximately the same for all thread counts, conveying that compiler instrumentation does not impact STM scalability. Table 4 summarizes costs introduced by the compiler instrumentation.

[2] The data we present here does not include STMBench7 workloads, due to the limitations of the STM compiler we used.

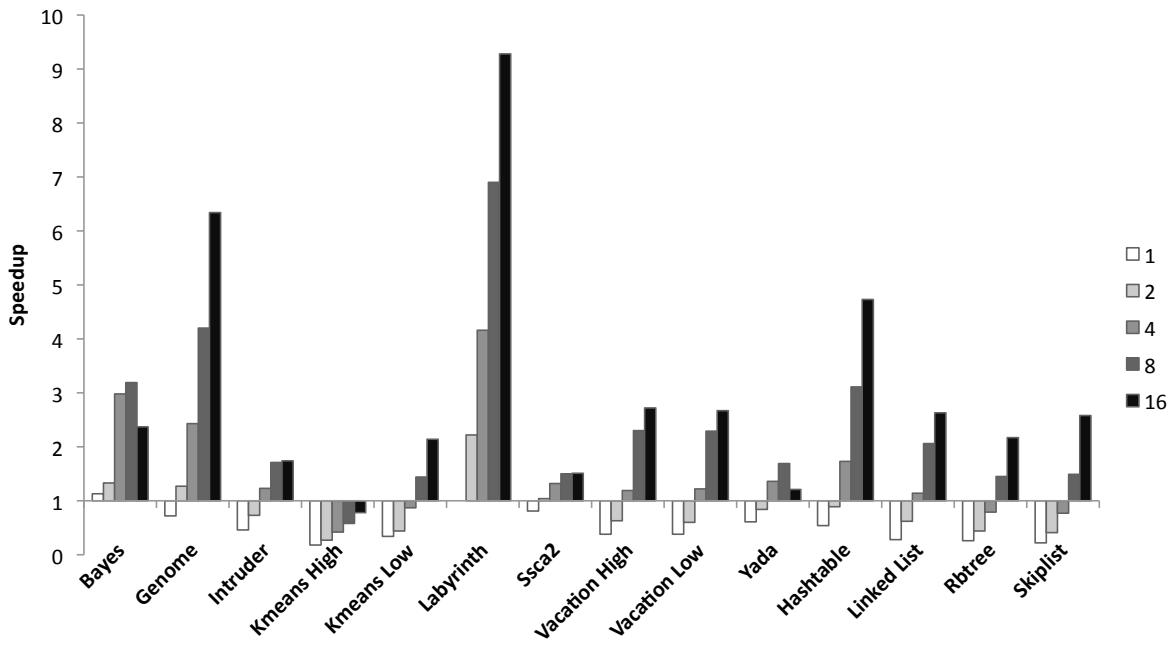


Figure 4. STM-CE performance with 16 core x86

To summarize, the additional overheads introduced by compiler instrumentation remain acceptable as STM-CE outperforms sequential code on 10 out of 14 workloads with only 4 threads and in all but one workload overall.

Further optimizations. In [18], 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 [22] to optimize the code generated by Tanger STM compiler.

Several optimizations have been proposed in the context of Java [1] to eliminate transactional accesses to immutable data and data allocated inside current transaction. Bartok-STM [14] uses flow-sensitive inter-procedural compiler analysis, as well as runtime log filtering, to identify objects allocated in the current transaction and eliminate transactional accesses to them. In [9] dataflow analysis is used to eliminate some unnecessary transactional accesses.

5.1 STM-CT Performance

We also performed experiments with STM-CT (STM using both compiler instrumentation and transparent privatization), but defer the result to the companion technical report [6] for lack of space. Our experiments show that, despite high costs of transparent privatization and compiler over-instrumentation, STM-CT performs well, outperforming sequential code in all but 4 workloads (out of 14). However, it requires higher thread counts to outperform sequential code than previous STM variants for the same workloads, as it outperforms sequential code in only 5 out of 14 workloads with 4 threads. The costs of STM-CT are largely a simple combination of STM-CE and STM-MT overheads and the same techniques for reducing transparent privatization and compilation overheads are applicable here.

5.2 Programming model

Our experiments imply that STM-CE (compiler instrumentation with explicit privatization) could be the most appropriate programming model for STM. 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. Clearly, an STM compiler is crucial for usability. Yet, transparent privatization support might not be absolutely needed from STM. It seems that privatizing a piece of data is a conscious decision made by a programmer rather than an accident. This might imply that explicitly marking privatizing transactions would not require too much additional effort from the programmer. Apart from semantic issues, our experiments show that STM-CE offers good performance and scales well .[3]

[3] The experiments in [3] were conducted with STM variants not supporting transparent privatization and, thus, this observation does not alter the performance comparisons we made in previous sections.

6. Conclusion

We reported on the most, to date, exhaustive evaluation of the ability of an STM to outperform sequential code. We showed that STM can do so across a wide range of workloads and different multicore architectures. Whereas we do not argue that STM is a silver bullet for general purpose concurrent programming, our results contradict [3] and suggest that STM is already now usable for various types of applications. These results support the initial hopes about STM performance and motivate further research in the field.

Many improvements could boost the STM performance further, making it even more appealing. 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, whilst reduction of accesses to shared data can enhance scalability.

Acknowledgments

We are grateful to Tim Harris for running the Bartok-STM experiments, to Calin Cascaval for providing us with the experimental settings of [3] and to Yang Ni for running experiments with McRT-STM confirming 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 their comments.

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

References

[1] 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*.

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

[3] 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), 2008.

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

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

[6] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. Technical Report LPD-REPORT-2009-003, 2009.

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

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

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

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

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

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

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

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

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

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

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

[18] 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*.

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

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

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

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

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

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

[25] 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*.