DOI:10.1145/1924421.1924440

Despite earlier claims, Software Transactional Memory outperforms sequential code.

BY ALEKSANDAR DRAGOJEVIĆ, PASCAL FELBER, VINCENT GRAMOLI, AND RACHID GUERRAOUI

Why STM Can Be More Than A Research Toy

WHILE MULTICORE ARCHITECTURES are increasingly the norm in CPUs, concurrent programming remains a daunting challenge for many. The transactional-memory paradigm simplifies concurrent programming by enabling programmers to focus on high-level synchronization concepts, or atomic blocks of code, while ignoring low-level implementation details.

Hardware transactional memory has shown promising results for leveraging parallelism[4] but is restrictive, handling only transactions of limited size or requiring some system events or CPU instructions to be executed outside transactions.[4] Despite attempts to address these issues,[19] TM systems fully implemented in hardware are unlikely to be commercially available for at least the next few years. More likely is that future deployed TMs will be hybrids containing a software component and a hardware component.

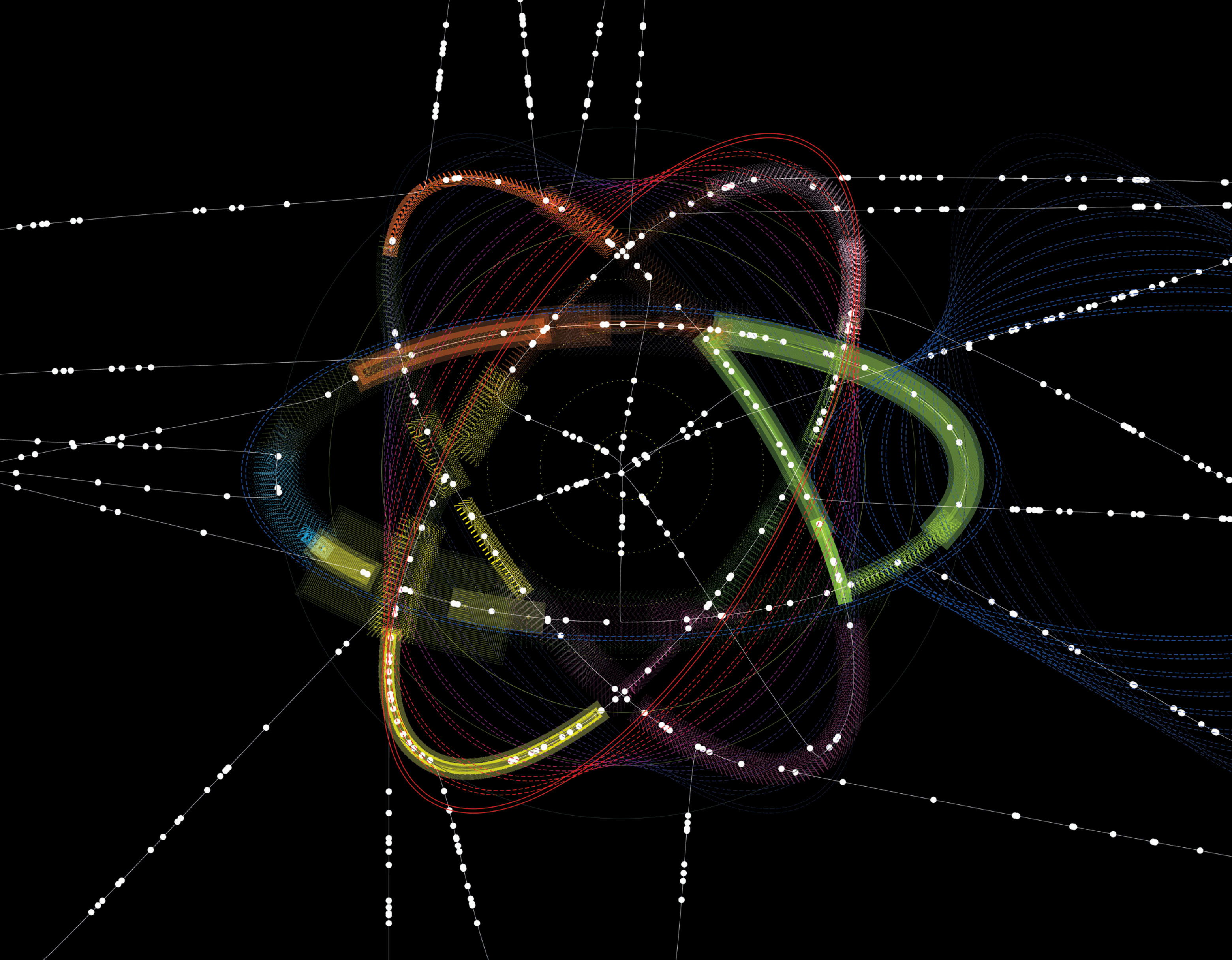
Software transactional memory[15,23] circumvents the limitations of HTM by implementing TM functionality fully in software. Moreover, several STM implementations are freely available and appealing for concurrent programming.[1,5,7,11,14,16,20] Yet STM credibility depends on the extent to which it enables application code to leverage multicore architectures and outperform sequential code. Cascaval et al.'s 2008 article[3] questioned this ability and suggested confining STM to the status of "research toy." STMs indeed introduce significant runtime overhead:

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, this access is performed by a single CPU instruction. STM read and write routines are significantly more expensive than corresponding CPU instructions, as they typically "bookkeep" data about every access. STMs check for conflicts, log 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 metadata, further increasing their cost.

Compiler overinstrumentation. Using

» key insights

- **STM is improving in terms of performance, often outperforming sequential, nontransactional code, when running with just four CPU cores.**
- **Parallel applications exhibiting high contention are not the primary target for STM so therefore are not the best benchmarks for evaluating STM performance.**
- **STM with support for compiler instrumentation and explicit, nontransparent privatization outperforms sequential code in all but one workload we used and still supports the programming model that is easy to use by typical programmers.**



an STM, programmers must 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, in which case a programmer manually replaces all memory references with STM calls or else lets it be performed by an STM compiler. With compilers, a programmer needs to specify only which sequences of statements must be atomic by enclosing them in transactional blocks. The compiler generates code invoking appropriate STM read/write calls. While an STM compiler significantly reduces programming complexity, it can also degrade the performance of resulting programs (compared to manual instrumentation) due to overinstrumentation[3,8,25]; basically, the compiler instruments the code conservatively with unnecessary calls to STM functions, as it cannot precisely determine which instructions access shared data.

Transparent privatization. Making certain shared data private to a certain thread, or "privatization," is typically used to allow nontransactional accesses to some data, to either support legacy code or improve performance by avoiding the cost of STM calls when accessing private data. Unless certain precautions are taken, privatization can result in race conditions.[24] Two approaches to prevent them: Either a programmer *explicitly* marks transactions that privatize data or STM *transparently* ensures all transactions safely privatize data. Explicit privatization puts additional burden on the programmer, while transparent privatization incurs runtime overhead,[25] especially when no data is actually privatized.

Many research papers[1–3,5,7,13,16,18,20] have discussed the scalability of STM with increasing numbers of threads, but few compare STM to sequential code or address whether STM is a viable option for speeding execution of applications.

Two notable exceptions are Minh et al.[2] and Cascaval et al.[3] In the former, STM was shown on a hardware simulator to outperform sequential code in most STAMP[2] benchmarks. The latter reported on a series of experiments on real hardware where STM performed worse than sequential code and even implied by the article's title that STM is only a research toy. A close look at the experiments reveal that Cascaval et al. considered a subset of the STAMP benchmark suite configured in a specific manner while using up to only eight threads.

We have since gone a step further, comparing STM performance to sequential code using a larger and more diverse set of benchmarks and real hardware supporting higher levels of concurrency. Specifically, we experimented with a state-of-the-art STM implementation, SwissTM[7] running three different STMBench7[12] workloads, all 10 workloads of the STAMP (0.9.10)[2] benchmark suite, and four microbenchmarks, 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 rest of this article) supporting 64 hardware threads and a four quad-core AMD Opteron x86 CPU machine (referred to as x86 in the rest of this article) supporting 16 hardware threads. Finally, we also considered all combinations of privatization and compiler support for STM (see Table 1). This constitutes the most exhaustive performance comparison of STM to sequential code published to date.

The experiments in this article (summarized in Table 2) show that STM does indeed outperform sequential code in most configurations and benchmarks, offering a viable paradigm for concurrent programming; 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 nine times on x86 with 16 concurrent threads. More important, STM performs well with a small number of threads on many benchmarks; for example, STM-ME outperforms sequential code with four threads on 14 of 17 workloads on our SPARC machine and on 13 of 17 workloads on our x86 machine. Basically, these results support early hope about the good performance of STM and should motivate further research. Our results contradict the results of Cascaval et al.[3] for three main reasons:

- STAMP workloads in Cascaval et al.[3] presented higher contention than default STAMP workloads;
- We used hardware supporting more threads and in case of x86 did not use hyperthreading; and
- We used a state-of-the-art STM implementation more efficient than those used in Cascaval et al.[3]

Table 1. STM support.

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

Clearly, and despite rather good STM performance in our experiments, there is room for improvement, and we use this article to highlight promising directions. Also, while use of STM involves several programming challenges[3] (such as ensuring weak or strong atomicity, semantics of privatization, and support for legacy binary code), alternative concurrency programming approaches (such as fine-grain locking and lock-free techniques) are no easier to use than STM. Such a comparison was covered previously[11,13,15,23] and is beyond our scope here.

Evaluation Settings

We first briefly describe the STM library used for our experimental evaluation, SwissTM,[7] along with benchmarks and hardware settings. Note that our experiments with other state-of-the-art STMs[5,14,18,20] on which we report in the companion technical report,[6] confirm the results presented here; SwissTM and the benchmarks are available at http://lpd.epfl.ch/site/research/tmeval.

The STM we used in our evaluation reflects three main features:

Synchronization algorithm. SwissTM[7] is a word-based STM that uses invisible (optimistic) reads, relying on a time-based scheme to speed up read-set validation, as in Dice et al.[5] and Riegel et al.[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 chosen to provide good performance across a range of workloads[7];

Privatization. We implemented privatization support in SwissTM using a simple validation-barriers scheme described in Spear et al.[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*; and

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

We conducted our experiments using the following benchmarks:

STMBench7. STMBench7[12] is a synthetic STM benchmark that models realistic large-scale CAD/CAM/CASE workloads, defining three different workloads with different amounts of contention: read-dominated (10% write operations), read/write (60% write operations), and write-dominated (90% write operations). The main characteristics of STMBench7 are a large data structure and long transactions compared to other STM benchmarks. In this sense, STMBench7 is very challenging for STM implementations;

STAMP. Consisting of eight different applications representative of

a Intel's C/C++ STM compiler generates only x86 code so was not used in our experiments on SPARC.

Table 2. Summary of STM speedup over sequential code.

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**

real-world workloads, STAMP[2] offers a range of workloads and is widely used to evaluate TM systems. STAMP applications can be configured with different parameters defining different workloads. In our experiments, we used 10 workloads from the STAMP 0.9.10 distribution, including 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]; and

Microbenchmarks. To evaluate low-level overhead of STMs (such as the cost of synchronization and logging) with smaller-scale workloads, we used four microbenchmarks 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 were filled with 2^{16} elements randomly chosen from a range of 2^{17} values. During the experiments, 5% of the transactions were insert operations, 5% were remove operations, and 90% were search operations.

It is important to note that these benchmarks are TM benchmarks, most using 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 we used. In a sense, the benchmarks we used represent a worst-case scenario for STM usage. For each experiment, we computed averages from at least five runs.

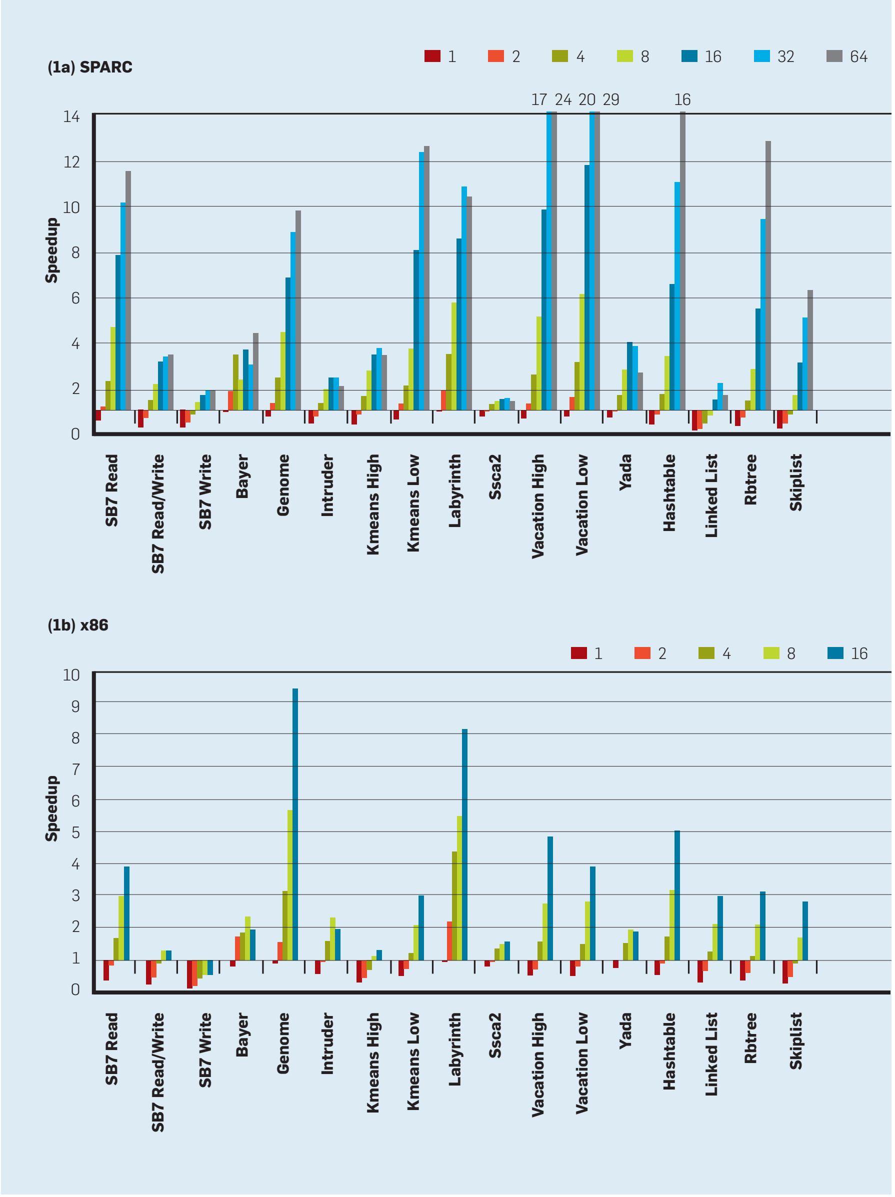
STM-ME Performance

Figure 1a outlines STM-ME (manual instrumentation with explicit privatization) speedup over sequential, noninstrumented code on SPARC, showing that STM-ME delivers good performance with a small number of threads, outperforming sequential code on 14 of 17 workloads with four threads. The figure also outlines that STM outperforms sequential code on all benchmarks we used by up to 29 times on the `vacation low` benchmark. The experiment shows that the less contention the workload exhibits, the more benefit is expected from STM; for example, STM outperforms sequential code by more than 11 times on a read-dominated workload of STMBench7 and less than two times for a write-dominated workload of the same benchmark.

On x86 (see Figure 1b), STM-ME outperforms sequential code on 13 workloads with four threads. Overall, STM clearly outperforms sequential code on all workloads, except on the challenging, high-contention `STMBench7 write` workload. The performance gain, compared to sequential code, is lower than on SPARC (up to nine times speedup on x86 compared to 29 times on SPARC) for two reasons: All threads execute on the same chip with SPARC, so the inter-thread communication costs less, and sequential performance of a single thread on SPARC is much lower.

STM-ME delivers good performance on both SPARC and x86 architectures, clearly showing STM-ME algorithms scale and perform well in different settings. However, also important is that, while STM-ME outperforms sequential code on all the benchmarks, some achieved speedups are not very impressive (such as 1.4 times with 64 threads on the `ssca2` benchmark). These lower speedups confirm that STM, despite showing great promise for some types of concurrent workloads, is not the best solution for all concurrent workloads.

Figure 1. STM-ME performance.



Contradicting earlier results. The results reported by Cascaval et al.[3] indicated STMs do not perform well on three of the STAMP applications we also used: `kmeans`, `vacation`, and `genome`. In our experiments, STM delivered good performance on all three. Three main reasons for such considerable difference are:

Workload characteristics. A close look at the experimental settings in Cascaval et al.[3] reveals 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, as in Figure 1.

To evaluate the impact of workload characteristics, we ran both default STAMP workloads and STAMP workloads from Cascaval et al.[3] on a machine with two quad-core Xeon CPUs that was more similar to the machine in Cascaval et al.[3] than to the x86 machine we used in our earlier experiments. Figure 2a outlines slowdown of workloads from Cascaval et al.[3] compared to default STAMP workloads; we used both low- and high-contention workloads for `kmeans` and `vacation`. Workload settings from Cascaval et al.[3] do indeed degrade STM-ME performance. The performance impact is significant in `kmeans` (around 20% for high- and up to 200% for low-contention workloads) and in `vacation` (30% to 50% in both). The performance is least affected in `genome` (around 10%).

Different hardware. We used hardware configurations with support for more hardware threads—64 and 16 in our experiments—compared to eight in Cascaval et al.[3] This significant difference lets STM perform better, as there is more parallelism at the hardware level to be exploited.

Our x86 machine does not use hyperthreading, while the one used by Cascaval et al.[3] does; hardware-thread multiplexing in hyperthreaded CPUs can hamper performance. To evaluate the effect, we ran the default STAMP workloads on a machine with two single-core hyperthreaded Xeon CPUs. Figure 2b outlines the slowdown on a hyperthreaded machine compared to a similar machine without hyperthreading. The figure shows that hyperthreading has a significant effect on performance, especially with higher thread counts. Slowdown in `genome` with four threads is around 65% and on two `vacation` workloads around 40%. The performance difference in `kmeans` workloads is significant, even with a single thread, due to differences in CPUs not related to hyperthreading. Still, even with `kmeans`, slowdown with four threads is much higher than with one and two threads.

More-efficient STM. Part of the performance difference is due to a more efficient STM implementation. The results reported by Dragojević et al.[7] suggest that SwissTM has better performance than TL2, performing comparably to the IBM STM in Cascaval et al.[3]

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

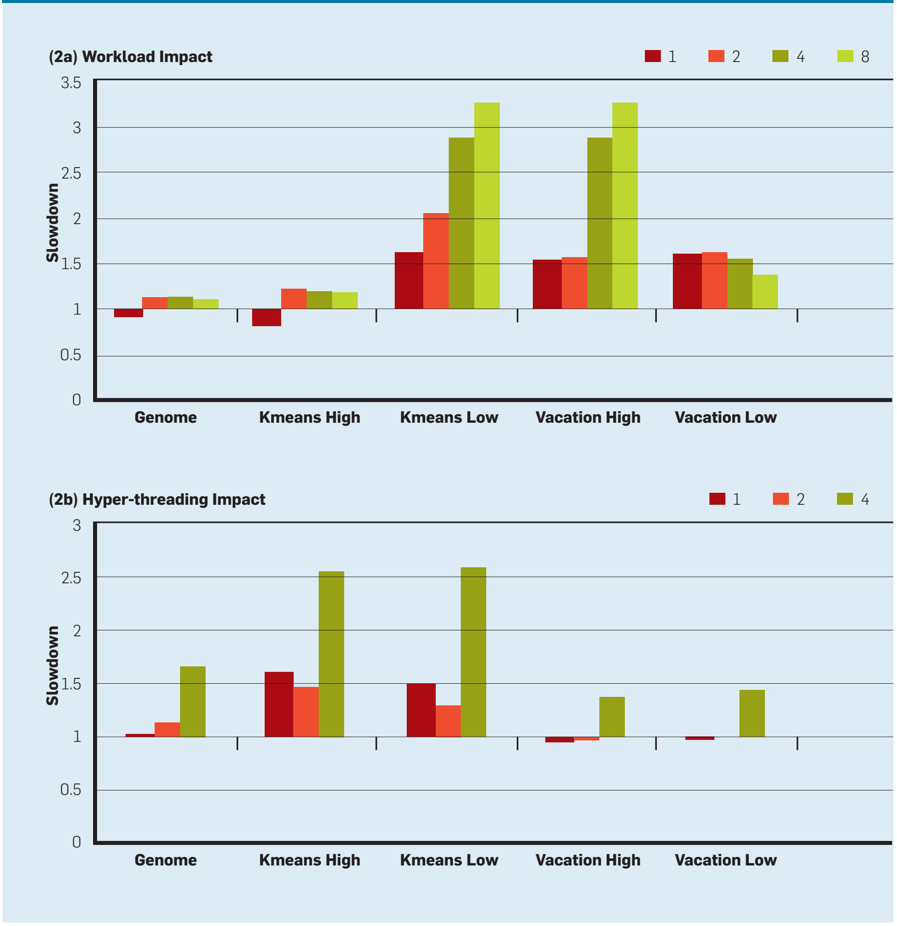
Further optimizations. In some workloads, performance degraded when we used too many concurrent threads. One possible alternative to improving performance in these cases would be to modify the thread scheduler so it avoids running more concurrent threads than is optimal for a given workload, based on the information provided by the STM runtime.

STM-MT Performance

Validation barriers we use to ensure privatization safety require frequent communication among all threads in the system and can degrade performance due to the time threads spend waiting for one another and the increased number of cache misses. A similar technique is known to significantly affect performance of STM in certain cases,[25] as confirmed by our experiments.

We must highlight the fact that none of the benchmarks we used requires privatization. We thus measured the worst case: Supporting transparent privatization incurs overhead without the performance benefits of reading and writing privatized data outside

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



of transactions. Also, the measured performance costs are specific to our choice of privatization technique and implementation; ways to reduce privatization costs have been proposed.[16,17]

Figure 3a shows performance of STM-MT (manual instrumentation with transparent privatization) with SPARC, conveying that transparent privatization affects the performance of STM significantly but that STM-MT still performs well, outperforming sequential code on 11 of 17 workloads with four threads and on 13 workloads with eight threads. Also, STM-MT outperforms sequential code on all benchmarks. However, performance is not as good; 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. Our experiments show performance for some workloads (such as `ssca2`) is unaffected but also that privatization costs can be as high as 80% (such as in `vacation low` and `yada`). Also, in general, costs increase with the number of concurrent threads, affecting both performance and scalability of STM. Table 3 summarizes the costs of transparent privatization with SPARC.

We repeated the experiments with the x86 machine (see Figure 3b), with results confirming that STM-MT has lower performance than STM-ME. STM-MT outperforms sequential code on eight of 17 workloads with four threads and on 14 workloads with eight threads. Overall, transparent privatization overhead reduces STM performance below performance of sequential code in three benchmarks: `STMBench7 read/write`, `STMBench7 write`, and `kmeans high`. Note that performance is affected most with microbenchmarks due to cache contention for shared privatization metadata induced by small transactions.

Our experiments show that privatization costs can be as high as 80% and confirm that transparent privatization costs increase with the number of threads. The cost of transparent privatization is higher on our four-CPU x86 machine than on SPARC, due mainly to the higher costs of interthread communication; Table 3 lists the costs of transparent privatization with x86.

While the effect of transparent privatization can be significant, STM-MT still scales and performs well on a range of applications. We also conclude that reducing costs of cache-coherence traffic by having more cores on a single chip reduces the cost of transparent privatization, resulting in better performance and scalability.

Further optimizations. Two recent proposals[16,17] aim to improve scalability of transparent privatization by employ-

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

	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**	—	—	—

Figure 3. STM-MT performance.

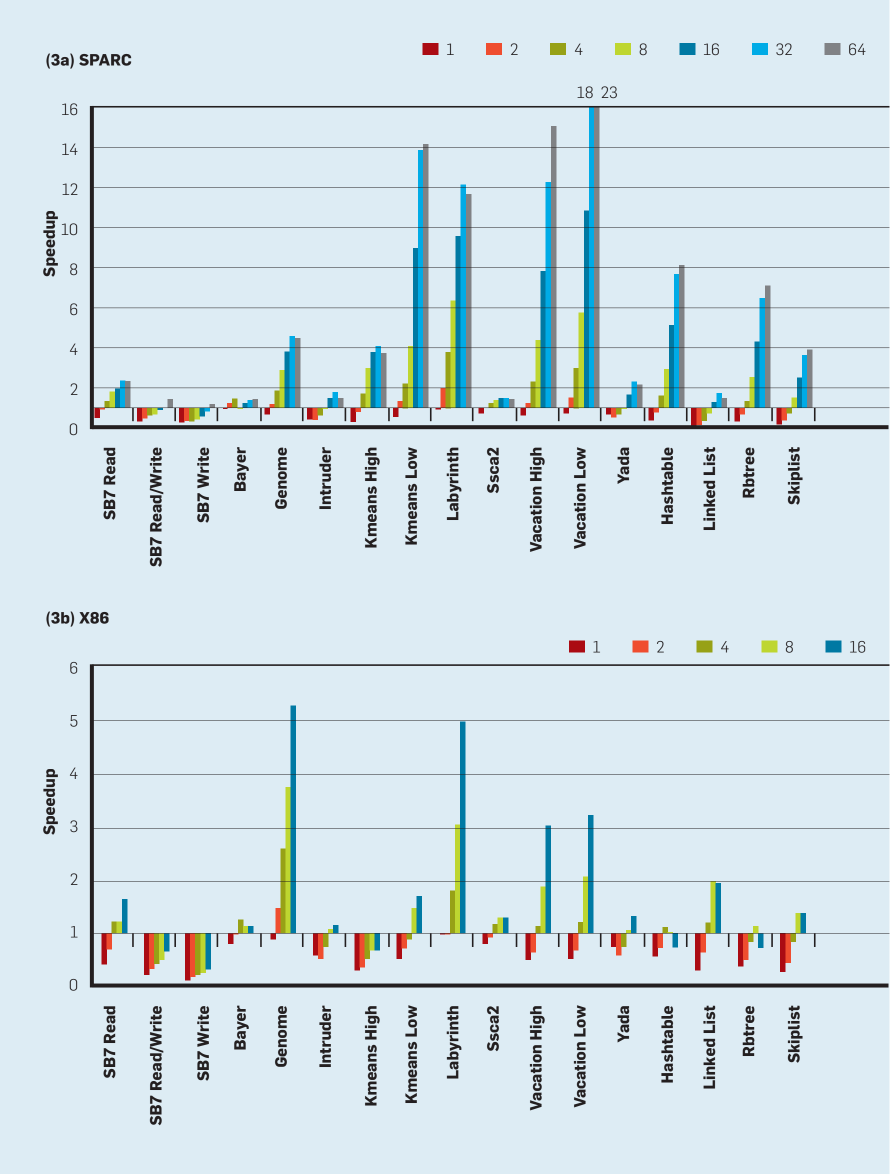


Table 4. Compiler instrumentation cost with x86 $(1 - \frac{\text{speedup}_{STM\text{-}CE}}{\text{speedup}_{STM\text{-}ME}})$.

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**

ing partially visible reads. By making readers only partially visible, the cost of reads is reduced, compared to fully visible reads, while improving the scalability of privatization support. To implement partially visible readers, Marathe et al.[17] used timestamps, while Lev et al.[16] used a variant of SNZI counters.[10] In addition, Lev et al.[16] avoided use of centralized privatization metadata to improve scalability.

STM-CE Performance

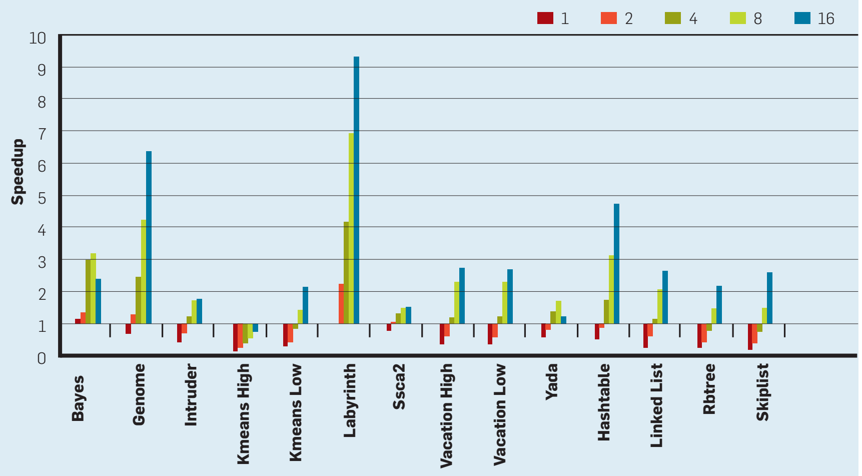
Compiler instrumentation often replaces more memory references by STM load and store calls than is strictly necessary, resulting in reduced performance of generated code, or "over-instrumentation."[3,8,25] Ideally, the compiler replaces only memory accesses with STM calls when they reference some shared data. However, the compiler does not have information about all uses of variables in the whole program or semantic information about variable use typically available only to the programmer (such as which variables are private to some threads and which are read-only). For this reason, the compiler, conservatively, generates more STM calls than necessary; unnecessary STM calls reduce performance because they are more expensive than the CPU instructions they replace.

Figure 4 outlines STM-CE (compiler instrumentation with explicit privatization) speedup over sequential code,[b] showing that STM-CE has good performance, outperforming sequential code on 10 of 14 workloads with four threads and on 13 workloads with eight threads. Overall, STM-CE outperforms sequential code in all benchmarks but `kmeans high` though scales well on `kmeans high`, promising to outperform sequential code with additional hardware threads.

The cost of compiler instrumentation is about 20% for all workloads but `kmeans`, where it is about 40%. Also, on some workloads (such as `labyrinth`, `ssca2`, and `hashtable`), compiler instrumentation does not introduce significant costs, and the performance of STM-ME and STM-CE is almost the same. In our experiments the costs of compiler instrumentation are approximately the same for all thread counts, conveying that compiler instrumentation does not affect STM scalability; Table 4 summarizes costs introduced by compiler instrumentation. The additional overheads introduced by compiler instrumentation remain acceptable, as STM-CE outperforms sequential code on 10 of 14 workloads with only four threads and on all but one workload overall.

b The data presented 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.



Further optimizations. Ni et al.[18] described optimizations that replace full STM load and store calls with specialized, faster versions of the same calls; for example, some STMs perform fast reads of memory locations previously accessed for writing inside the same transaction. While the compiler we used supports these optimizations, we have not yet implemented the lower-cost STM barriers in SwissTM. Compiler data structure analysis was used by Riegel et al.[22] to optimize the code generated by the Tanger STM compiler.

Adl-Tabatabai et al.[1] proposed several optimizations in the Java context to eliminate transactional accesses to immutable data and data allocated inside current transactions. Harris et al.[14] used flow-sensitive interprocedural compiler analysis, as well as runtime log filtering in Bartok-STM, to identify objects allocated in the current transaction and eliminate transactional accesses to them. Eddon and Herlihy[9] used dataflow analysis of Java programs to eliminate some unnecessary transactional accesses.

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

Programming model. The experiments we report here imply that STM-CE (compiler instrumentation with explicit privatization) may be the most appropriate programming model for STM. STM-ME is likely too tedious and error-prone for use in most applications and might instead be appropriate only for smaller applications or performance-critical sections of code. Clearly, an STM compiler is crucial for usability, yet transparent privatization support might not be absolutely needed from STM. It seems that programmers make a conscious decision to privatize a piece of data, rather than let the data be privatized by accident. This might imply that, for the programmer, explicitly marking privatizing transactions would not require much additional effort. Apart from semantic issues, our experiments show that STM-CE offers good performance while scaling well.[c]

Conclusion

We reported on the most exhaustive evaluation to date of the ability of an STM to outperform sequential code, showing it can deliver good performance across a range of workloads and multicore architectures. Though we do not claim STM is a silver bullet for general-purpose concurrent programming, our results contradict Cascaval et al.[3] and suggest STM is usable for a range of applications. They also support the initial hopes about STM performance and should motivate further research in the field.

Many improvements promise to boost STM performance further, making it even more appealing; for example, static segregation of memory locations, depending on whether or not the locations are shared, 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, to Calin Cascaval for providing us with the experimental settings of Cascaval et al.,[3] and to Yang Ni for running experiments with McRT-STM, confirming our speculation about the effect of hyperthreading. We also thank Hillel Avni, Derin Harmanci, Michał Kapałka, Patrick Marlier, Maged Michael, and Mark Moir for their comments. This work was funded by the Velox FP7 European Project and the Swiss National Science Foundation grant 200021-116745/1. C

c Because the experiments reported in Cascaval et al.[3] were conducted with STM variants not supporting transparent privatization, our observation about the programming model does not alter the performance comparisons covered earlier in the article.

References

1. Adl-Tabatabai, A.-R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, June 10–16). ACM Press, New York, 2006, 26–37.
2. Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. STAMP: Stanford Transactional Applications for Multiprocessing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization* (Seattle, Sept. 14–16). IEEE Computer Society, Washington, D.C., 2008, 35–46.
3. Cascaval, C., Blundell, C., Michael, M.M., Cain, H.W., Wu, P., Chiras, S., and Chatterjee, S. Software transactional memory: Why is it only a research toy? *Commun. ACM 51*, 11 (Nov. 2008), 40–46.
4. Dice, D., Lev, Y., Moir, M., and Nussbaum, D. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, D.C., Mar. 7–11). ACM Press, New York, 2009, 157–168.
5. Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing* (Stockholm, Sept. 18–20). Springer-Verlag, Berlin, 2006, 194–208.
6. Dragojević, A., Felber, P., Gramoli, V., and Guerraoui, R. *Why STM Can Be More Than a Research Toy. Technical Report LPD-REPORT-2009-003.* EPFL, Lausanne, Switzerland, 2009.
7. Dragojević, A., Guerraoui, R., and Kapalka, M. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, 2009). ACM Press, New York, 2009, 155–165.
8. Dragojević, A., Ni, Y., and Adl-Tabatabai, A.-R. Optimizing transactions for captured memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures* (Calgary, Aug. 11–13). ACM Press, New York, 2009, 214–222.
9. Eddon, G. and Herlihy, M. Language support and compiler optimizations for STM and transactional boosting. In *Proceedings of the Fourth International Conference on Distributed Computing and Internet Technology* (Bangalore, Dec. 17–20). Springer-Verlag, Berlin, 2007, 209–224.
10. Ellen, F., Lev, Y., Luchangco, V., and Moir, M. Snzi: Scalable nonzero indicators. In *Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Portland, OR, Aug. 12–15). ACM Press, New York, 2007, 13–22.
11. Felber, P., Gramoli, V., and Guerraoui, R. Elastic transactions. In *Proceedings of the 23rd International Symposium on Distributed Computing* (Elche/Elx, Spain, Sept. 23–25). Springer-Verlag, Berlin, 2009, 93–107.
12. Guerraoui, R., Kapalka, M., and Vitek, J. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems* (Lisbon, Mar. 21–23). ACM Press, New York, 2007, 315–324.
13. Harris, T. and Fraser, K. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA, Oct. 26–30). ACM Press, New York, 2003, 388–402.
14. Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, June 10–16). ACM Press, New York, 2006, 14–25.
15. Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W.N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing* (Boston, July 13–16). ACM Press, New York, 2003, 92–101.
16. Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., and Olszewski, M. Anatomy of a scalable software transactional memory. In *Proceedings of the Fourth ACM SIGPLAN Workshop on Transactional Computing* (Raleigh, NC, Feb. 15, 2009).
17. Marathe, V.J., Spear, M.F., and Scott, M.L. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 37th International Conference on Parallel Processing* (Portland, OR, Sept. 8–12). IEEE Computer Society, Washington, D.C., 2008, 67–74.
18. Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., and Tian, X. Design and implementation of transactional constructs for C/C++. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Nashville, Oct. 19–23). ACM Press, New York, 2008, 195–212.
19. Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Madison, WI, June 4–8). IEEE Computer Society, Washington, D.C., 2005, 494–505.
20. Riegel, T., Felber, P., and Fetzer, C. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, Feb. 20–23). ACM Press, New York, 2008, 237–246.
21. Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing* (Stockholm, Sept. 18–20). Springer-Verlag, Berlin, 2006, 284–298.
22. Riegel, T., Fetzer, C., and Felber, P. Automatic data partitioning in software transactional memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Münich, June 14–16). ACM Press, New York, 2008, 152–159.
23. Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Aug. 20–23). ACM Press, New York, 1995, 204–213.
24. Spear, M.F., Marathe, V.J., Dalessandro, L., and Scott, M.L. Privatization techniques for software transactional memory. In *Proceedings of the 26th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Portland, OR, Aug. 12–15). ACM Press, New York, 2007, 338–339; extended version available as TR 915, Computer Science Department, University of Rochester, Feb. 2007.
25. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.-R., and Lee, H.H.S. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Münich, June 14–16). ACM Press, New York, 2008, 265–274.

Aleksandar Dragojević (aleksandar.dragojevic@epfl.ch) is a Ph.D. student in the Distributed Programming Laboratory of the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

Pascal Felber (pascal.felber@unine.ch) is a professor in the Computer Science Department of the Université de Neuchâtel, Neuchâtel, Switzerland.

Vincent Gramoli (vincent.gramoli@epfl.ch) is a postdoctoral researcher in the Distributed Computing Laboratory in the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, and a postdoctoral researcher in the Computer Science Department of the Université de Neuchâtel, Neuchâtel, Switzerland.

Rachid Guerraoui (rachid.guerraoui@epfl.ch) is a professor in the Distributed Programming Laboratory in the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.