


software transactional memory

why is it
only a
research
toy?





Călin Cașcaval, Colin Blundell,
Maged Michael, Harold W. Cain,
Peng Wu, Stefanie Chiras,
and Siddhartha Chatterjee

The overhead posed by STM may likely overshadow its promise.

TM (transactional memory)¹ is a concurrency control paradigm that provides atomic and isolated execution for regions of code. TM is considered by many researchers to be one of the most promising solutions to address the problem of programming multicore processors. Its most appealing feature is that most programmers only need to reason locally about shared data accesses, mark the code region to be executed transactionally, and let the underlying system ensure the correct concurrent execution. This model promises to provide the scalability of fine-grain locking, while avoiding common pitfalls of lock composition such as deadlock. In this article we explore the performance of a highly optimized STM and observe that the overall performance of TM is significantly worse at low levels of parallelism, which is likely to limit the adoption of this programming paradigm.

Different implementations of transactional memory systems make tradeoffs that impact both performance and programmability. Larus and Rajwar² present an overview of design tradeoffs for implementations of transactional memory systems. Here are some of the design choices:

- STM (software-only TM)^{3,4,5,6,7,8,9} is the focus of this article. While offering flexibility and no hardware cost, it leads to overhead in excess of most users' tolerance.
- HTM (hardware-only TM)^{10,11,12,13,14,15,16} suffers from two major impediments: high implementation and verification costs lead to design risks too large to justify on a niche programming model; and hardware capacity constraints lead to significant performance

software transactional memory

→ why is it only a research toy?

degradation when overflow occurs, and proposals for managing overflows (for example, signatures¹⁷) incur false positives that add complexity to the programming model. Therefore, from an industrial perspective, HTM designs have to provide more benefits for the cost on a more diverse set of workloads (with varying transactional characteristics) for hardware designers to consider implementation. (Reuse of hardware for other purposes can also justify its inclusion, as may be the case for Sun's implementation of Scout Threading in the Rock processor.¹⁸)

- Hybrid systems^{19,20,21,22} are the most likely platform for the eventual adoption of TM by a wide audience, although the exact mix of hardware and software support remains unclear. A special case of the hybrid system is the hardware-accelerated STM. In this scenario, the transactional semantics are provided by STM, and hardware primitives are used only to speed up critical performance bottlenecks in the STM system. Such systems could offer an attractive solution if the cost of hardware primitives is modest and may be further amortized by other uses.

Independent of these implementation decisions, there are transactional semantics issues that break the ideal transactional programming model for which the community had hoped. TM introduces a variety of programming issues that are not present in lock-based mutual exclusion. For example, semantics are muddled by:

- Interaction with nontransactional codes, including access to shared data from outside of a transaction (tolerating weak atomicity) and the use of locks inside a transaction (breaking isolation to make locking operations visible outside transactions).
- Exceptions and serializability—how to handle exceptions and propagate consistent exception information from within a transactional context, and how to guarantee that transactional execution respects a correct ordering of operations.

- Interaction with code that cannot be transactionalized, as a result of either communication with other threads or a requirement barring speculation.

- Livelock, or the system guarantee that all transactions make progress even in the presence of conflicts.

In addition to the intrinsic semantic issues, there are also implementation-specific optimizations motivated by high transactional overheads, such as programmer annotations for excluding private data. Furthermore, the nondeterminism introduced by aborting transactions complicates debugging—transactional code may be executed and aborted on conflicts, which makes it difficult for the programmer to find deterministic paths with repeatable behavior. Both of these dilute the productivity argument for transactions, especially software-only TM implementations.

Given all these issues, we conclude that TM has not yet matured to the point where it presents a compelling value proposition that will trigger its widespread adoption. While TM can be a useful tool in the parallel programmer's portfolio, it is not going to solve the parallel programming dilemma by itself. There is evidence that it helps with building certain concurrent data structures, such as hash tables and binary trees. In addition, there are anecdotal claims that it helps with workloads; however, despite several years of active research and publication in the area, we are disappointed to find no mentions in the research literature of large-scale applications that make use of TM. The STAMP²³ (Stanford Transactional Applications for Multiprocessing) and Lonestar²⁴ benchmark suites are promising starts but have a long way to go to be representative of full applications.

We base these conclusions on our work over the past two years building a state-of-the-art STM runtime system and compiler framework, the freely available IBM STM.²⁵ Here, we describe this experience, starting with a discussion of STM algorithms and design decisions. We then compare the performance of this STM with two other state-of-the-art implementations (the Intel STM²⁶ and the Sun TL2 STM²⁷), as well as dissect the operations executed by the IBM STM and provide a detailed analysis of the performance hotspots of the STM.

SOFTWARE TRANSACTIONAL MEMORY

STM implements all the transactional semantics in software. That includes conflict detection, guaranteeing the consistency of transactional reads, preservation of atomicity and isolation (preventing other threads from observing speculative writes before the transaction succeeds), and conflict resolution (transaction arbitration).

The pseudocode for the main operations executed by a typical STM is illustrated in figure 1. It shows two STM algorithms: one that performs full validation and one that uses a global version number (the additional statements marked with the `gv#` comment).

The advantage of STM for system programmers is that it offers flexibility in implementing different mechanisms and policies for these operations. For end users, the advantage of STM is that it offers an environment to transactionalize (i.e., port to TM) their applications without incurring extra hardware cost or waiting for such hardware to be developed.

On the other hand, STM entails nontrivial drawbacks with respect to performance and programming semantics:

Overheads. In general, STM results in higher sequential overheads than traditional shared-memory programming or HTM. This is the result of the software expansion of loads and stores to shared mutable locations inside transactions to tens of additional instructions that constitute the STM implementation (for example, the `STM_READ` code in figure 1). Depending on the transactional characteristics of a workload, these overheads can become a high hurdle for STM to achieve performance. The

sequential overheads (that is, conflict-free overheads that are incurred regardless of the actions of other concurrent threads) must be overcome by the concurrency-enabling characteristics of transactional memory.

Semantics. To avoid incurring high STM overheads, nontransactional accesses (i.e., loads and stores occurring outside transactions) are typically not expanded. This has the effect of weakening—and hence complicating—the semantics of transactions, which may require the programmer to be more careful than when strong transactional semantics are supported. The following are some of the weakened guarantees that are usually associated with such STMs:

- **Weak atomicity.** Typically, the STM runtime libraries cannot detect conflicts between transactions and non-transactional accesses. Thus, the semantics of atomicity are weakened to allow undetected conflicts with non-transactional accesses (referred to as weak atomicity²⁸), or equivalently put the burden on the programmer to guarantee that no such conflicts can possibly take place.
- **Privatization.** Some STM designs prohibit the seamless privatization of memory locations (that is, the transition from being accessed transactionally to being accessed

FIGURE 1

STM Operations

Pseudo-code for STM begin

```
STM_BEGIN()
  read global version number /* gv# */
```

Pseudo-code for STM validate

```
STM_VALIDATE()
  read global version number /* gv# */
  if global version number changed /* gv# */
    for each read set entry
      if metadata changed return FALSE
  return TRUE
```

Pseudo-code for STM read barrier

```
STM_READ(A)
  if already written goto written path
  read metadata of A
  if metadata is locked goto conflict path
  log A and its metadata in the read set
  read value at A
  if ! STM_VALIDATE() goto conflict path
  return val
```

Pseudo-code for STM end

```
STM_END()
  lock metadata for write set
  if already locked goto conflict path
  if ! STM_VALIDATE() goto conflict path
  /* Success guaranteed */
  increment global version number /* gv# */
  execute writes
  update/unlock metadata for write set
```

software transactional memory

→ why is it only a research toy?

privately—or nontransactionally in general, by using locks). For some STM designs, once a location is accessed transactionally, it must continue to be accessed that way. Sometimes, the programmer can ease the transition by guaranteeing that the first access to the privatized location—such as after the location is no longer accessible by other threads—is transactional.

- **Memory reclamation.** Some STM designs prohibit the seamless reclamation of the memory locations accessed transactionally for arbitrary reuse, such as using malloc and free. With such STM designs, memory allocation and deallocation for locations accessed transactionally are handled differently than for other locations.

- **Legacy binaries.** STM needs to observe all memory activities of the transactional regions to ensure atomicity and isolation. STM designs that achieve this observation by code instrumentation generally cannot support transactions calling legacy codes that are not instrumented (for example, third-party libraries) without seriously limiting concurrency, such as by serializing transactions.

EVALUATION

Throughout this section we use the following set of benchmarks:

- **b+tree** is an implementation of database indexing operations on a b-tree data structure for which the data

FIGURE 2

Scalability Results for Three STM Runtimes on a Quad-Core Intel Xeon Server

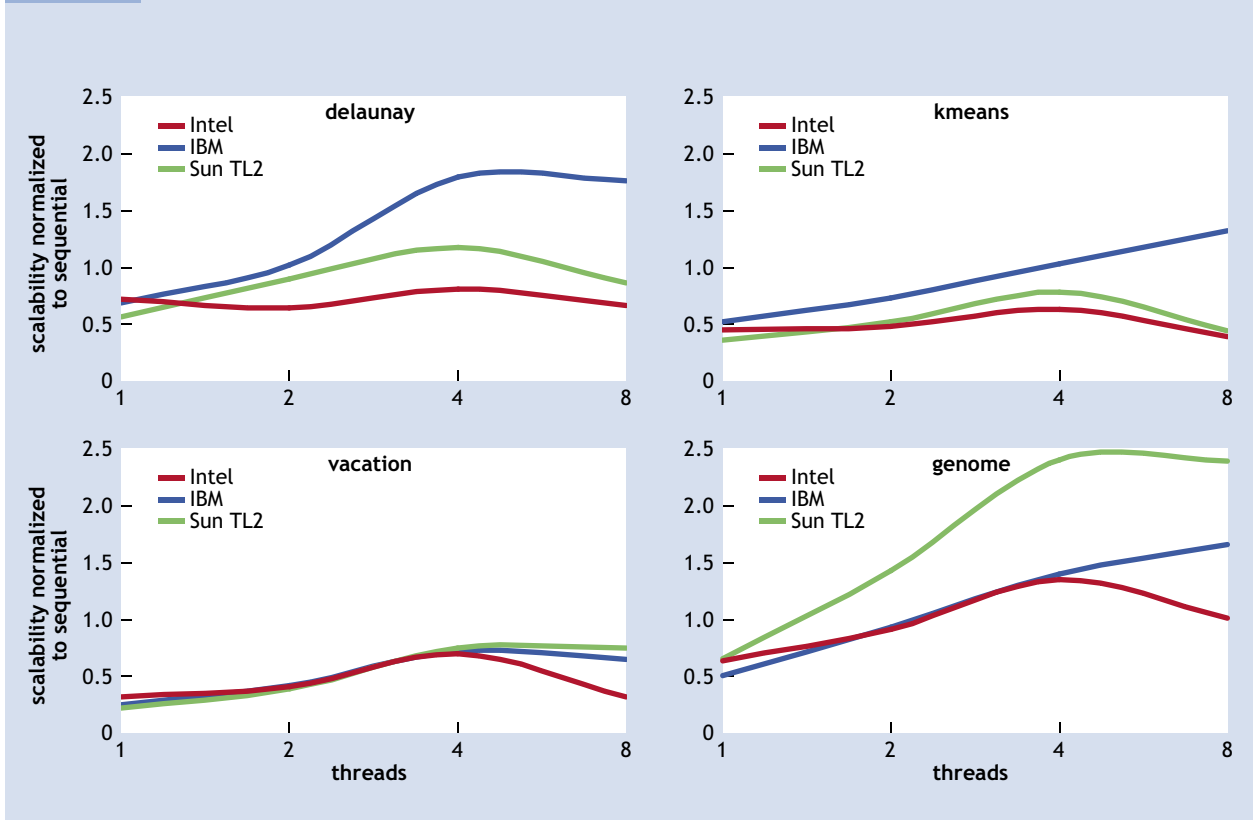
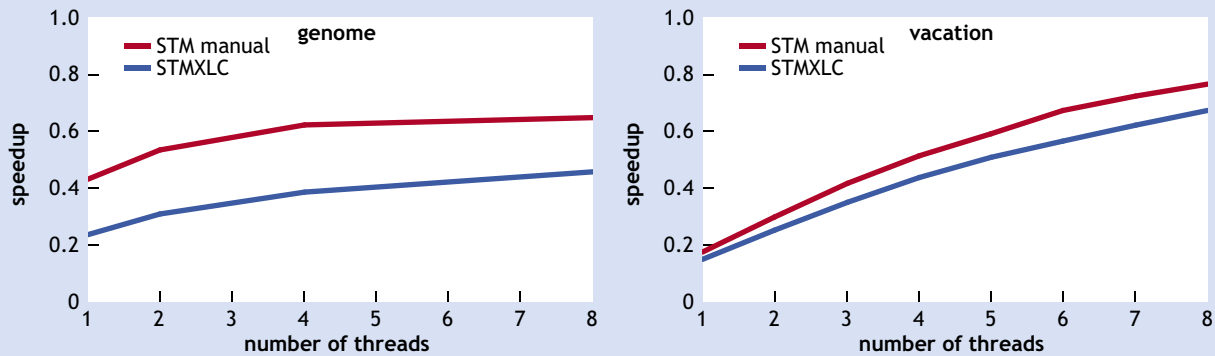


FIGURE 3

Scalability for Manual and Compiler Instrumented Benchmarks on AIX PowerPC



is stored only on the tree leaves (a b+ tree). This implementation uses coarse-grained transactions for every tree operation. Each b+ tree operation starts from the tree root and descends to the leaves. A leaf update may trigger a structural modification to rebalance the tree. A rebalancing operation often involves recursive ascent over the child-parent edges. In the worst case, the rebalancing operation modifies the entire tree. Our workload inserts 2,048 items in a b+ tree of order 20. For this code we have only a transactional version that is not manually instrumented; therefore, experimental results are presented only in configurations where we can use our compiler to provide instrumentation.

- *delaunay* implements the Delaunay Mesh Refinement algorithm described in Kulkarni et al.²⁹ The code produces a guaranteed quality Delaunay mesh. This is a Delaunay triangulation with the additional constraint that no angle in the mesh be less than 30 degrees. The benchmark takes as input an unrefined Delaunay triangulation and produces a new triangulation that satisfies this constraint. In the TM implementation of the algorithm, multiple threads choose their elements from a work queue and refine the cavities as separate transactions.
- *genome*, *kmeans*, and *vacation* are part of the STAMP benchmark suite³⁰ version 0.9.4. For a detailed description of these benchmarks, see STAMP.³¹

Baseline performance. Figure 2 presents a performance comparison of three STMs: IBM,^{32,33} Intel,³⁴ and Sun TL2.³⁵ The runs are on a quad-core, two-way hyperthreaded Intel Xeon 2.3-GHz box running Linux Fedora Core 6. In

these runs, we used the manually instrumented versions of the codes, which aggressively minimize the number of barriers for the IBM and TL2 STMs. Since we do not have access to low-level APIs for the Intel STM, its curves are from codes instrumented by the Intel STM compiler, which incurs additional barrier overheads as a result of compiler instrumentation.³⁶ The graphs are scalability curves with respect to the serial, nontransactionalized version. Therefore, a value of 1 on the y-axis represents performance equal to the serial version.

The performance of these STMs is mostly on par, with the IBM STM showing better scalability on *delaunay* and TL2 obtaining better scalability on *genome*. The overall performance obtained is very low, however: on *kmeans* the IBM STM barely attains single-threaded performance at four threads, while on *vacation* none of the STMs actually overcomes the overhead of transactional memory even with eight threads.

Compiler instrumentation. The compiler is a necessary component of an STM-based programming environment that is to be adopted by mass programmers. Its basic role is to eliminate the need for programmers to manually instrument memory references to STM read and write barriers. While offering convenience, compiler instrumentation does add another layer of overheads to the STM system by introducing redundant barriers, often resulting from the conservativeness of compiler analysis, as observed in Yoo.³⁷

software transactional memory

→ why is it only a research toy?

Figure 3 provides another baseline: the overhead of compiler instrumentation. The performance is measured on a 16-way POWER5 running AIX 5.3. For the STMXLC curve, we use the uninstrumented versions of the codes and annotate transactional regions and functions using the language extensions provided by the compiler.³⁸

Compiler over-instrumentation is more pronounced in traditional, unmanaged languages such as C and C++, where a typical compiler instrumentation without interprocedural analysis may end up instrumenting every memory reference in the transactional region (except for stack accesses). For example, our compiler instrumentation more than doubled the number of dynamic read barriers in *delaunay*, *genome*, and *kmeans*. Interprocedural analysis can help improve the tightness of compiler instrumentation for some cases but is generally limited by the accuracy of global analysis.

STM operations performance. Given this baseline, we now analyze in detail which operations in the STM cause the overhead. For this purpose, we use a cycle-accurate simulator of the PowerPC architecture that provides hooks for instrumentation. The STM operations and suboperations are instrumented with these simulator hooks. The reason for this

environment is that we want to capture the overheads at the instruction level and eliminate any other nondeterminism introduced by real hardware. The simulator eliminates all other bookkeeping operations introduced by instrumentation and provides an accurate breakdown of the STM overheads.

We study the performance of two STM algorithms: one that fully validates (fv) the read set after each transactional read and one that uses a global version number (gv#) to avoid the full validation, while maintaining the correctness of the operations. The fv algorithm provides more concurrency at a much higher price. The gv# is deemed as one of the best tradeoffs for STM implementations.

Figure 4 presents the single-threaded overhead of these algorithms over sequential runs, illustrating again the substantial slowdowns that the algorithms induce. Figure 5 breaks down these overheads into the various STM components. For both algorithms, the overhead of transactional reads dominates because of the frequency of read operations relative to all other operations. The effectiveness of the global version number in reducing overheads is shown in the lower read overhead of gv#.

FIGURE 4

Single-Threaded Overhead of the STM Algorithms

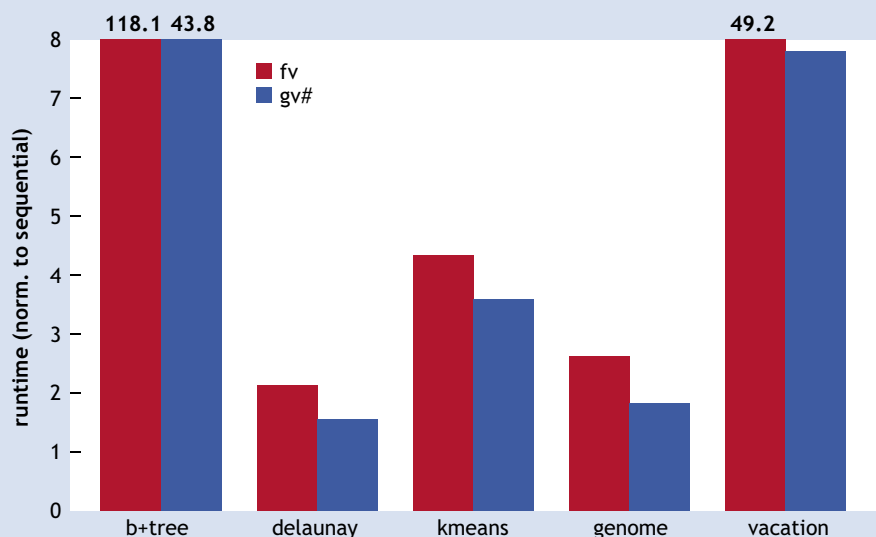


FIGURE 5

Percentage of Time Spent in Different STM Operations

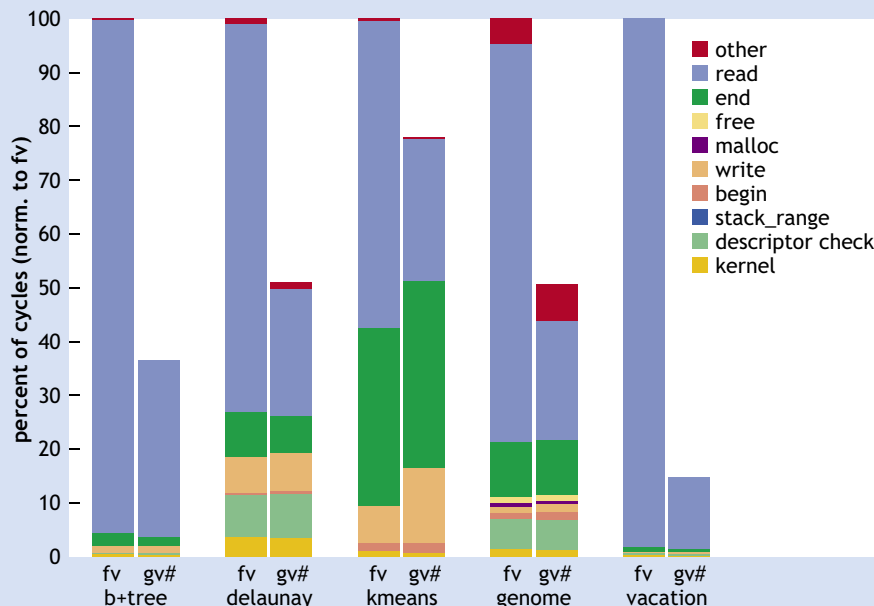


Figure 6 gives a fine-grained breakdown of the overheads of the transactional read operation. As expected, the overhead of validating the read set dominates transactional read time in the fv configuration. For both algorithms, the isync operations (necessary for ordering the metadata read and data read, as well as the data read and validation) form a substantial component. In applications that perform writes before reads in the same transaction (*delaunay*, *kmeans*), the time spent checking whether a location has been written by prior transactional writes in the same transaction forms a significant component of the total time. Interestingly, reading the data itself is a negligible amount of the total time, indicating the hurdles that must be overcome for the performance of these algorithms to be compelling.

Figure 7 gives a similar breakdown of the transactional commit operation. As before, the fv configuration suffers from having to validate the read set. Other dominant overheads for both configurations are those of having to acquire the metadata for the write set (which involves a sequence of load-linked/store-conditional operations) and

the sync operations that are necessary for ordering the metadata acquires, data writes, and metadata releases. Once again, the data writes themselves form a small component of the total time.

Overhead optimizations. There have been many proposals on reducing STM overheads through compiler or runtime techniques. Most of these techniques are complementary to hardware acceleration for STM.

- **Redundant barrier elimination.** One technique is to eliminate barriers to thread-local objects through *escape analysis*. Such analysis is typically quite effective in identifying thread-local accesses that are close to the object allocation site. It can eliminate both read and write

barriers but is often more effective on write barriers. For example, we observe that an intraprocedural escape analysis can eliminate 40 to 50 percent of write barriers in *vacation*, *genome*, and *b+tree*. Its impact on performance is more limited, however: from negligible to 12 percent. To target redundant read barriers, a whole-program analysis called Not-Accessed-In-Transaction³⁹ eliminates some barriers to read-only objects in transactions.

- **Barrier strength reduction.** These optimizations do not eliminate barriers but identify at runtime special locations that require only lightweight barrier processing, such as dynamic tracking of thread-local objects^{40,41} and runtime filtering of stack references and duplicate references.⁴²

- **Code generation optimizations.** One common technique is to inline the fast path of barriers. It has the potential benefit of reducing function-call overhead, increasing ILP, and exposing reuse of common sub-barrier operations. In our experiments, compiler inlining achieved less than 2 percent overall improvement across our benchmark suite.

software transactional memory

→ why is it only a research toy?

- **Commit sequence optimizations.** Eliminating unnecessary global version number updates⁴³ improves the overall performance of several micro-benchmarks by up to 14 percent.

Such optimizations have a positive impact on STM performance. The results presented here, however, indicate how much further innovation is needed for the performance of STM systems to become generally appealing to users.

RELATED WORK

The first STM system was proposed by Shavit and Touitou⁴⁴ and is based on object ownership. The protocol is static, which is a significant shortcoming that has been overcome by subsequently proposed STM systems.⁴⁵ Conflict detection is simplified significantly by the static nature because conflicts can be ruled out already when ownership records are acquired (at transaction start).

DSTM⁴⁶ is the first dynamic STM system; the design follows a per-object runtime organization (locator object). Variables (objects) in the application heap refer to a locator object. Unlike in a design with ownership records (for example, Harris and Fraser⁴⁷), the locator does

not store a version number but refers to the most recently committed version of the object. A particularity of the DSTM design is that objects must be explicitly opened (in read-only or read-write mode) before transactional access; also, DSTM allows for early release. The authors argue that both mechanisms facilitate the reduction of conflicts.

The design principles of the RSTM⁴⁸ (Rochester STM) system are similar to DSTM in that it associates transactional metadata with objects. Unlike DSTM, however, the system does not require the dynamic allocation of transactional data but colocates it with the nontransactional data. This scheme has two benefits: first, it facilitates spatial access locality and hence fosters execution performance and transaction throughput; second, the dynamic memory management of transactional data (usually done through a garbage collector) is not necessary, and hence this scheme is amenable to use in environments where memory management is explicit.

Recent work has explored algorithmic optimizations and/or alternative implementations of the basic STM algorithms described in this article. Riegel et al. propose the use of realtime clocks to enhance the scalability of

FIGURE 6

Percentage of Time Spent in STM Read Sub-Operations

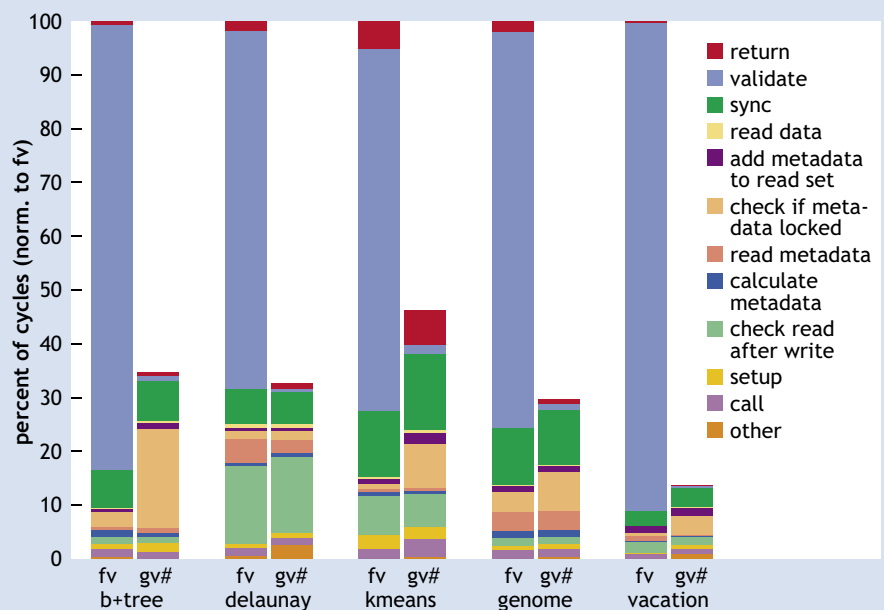
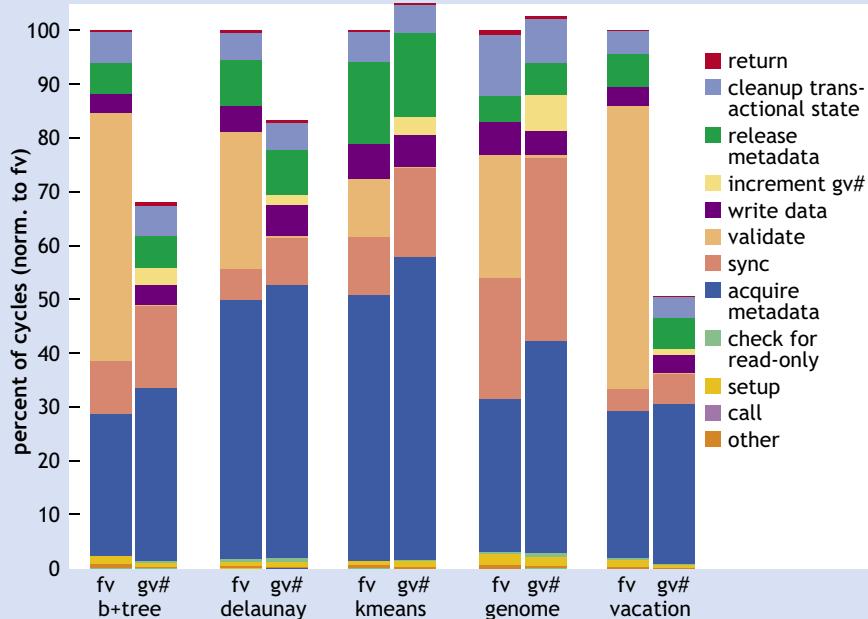


FIGURE 7

Percentage of Time Spent in STM End Sub-Operations



performance by 2 to 15 percent. We believe such analysis is a good practice that should be extended to every piece of system software, especially open source. However, the gains are only a minor dent in the overheads we observed, indicating the challenge that lies before the community in making STM performance compelling.

CONCLUSION

Based on our results, we believe that the road ahead for STM is quite challenging. Lowering the overhead of STM to a point where it is generally appealing is a difficult task, and significantly better results have to be demonstrated. If we could stress a single direction for further research, it is the elimination of dynamically unnecessary read and write barriers—

STMs that use a global version number.⁴⁹ JudoSTM⁵⁰ and RingSTM⁵¹ reduce the number of atomic operations that must be performed when committing a transaction at the cost of serializing commit and/or incurring spurious aborts because of imprecise conflict detection. Several proposals have been made for STM systems that operate via dynamic binary rewriting in order to allow the usage of STM on legacy binaries.^{52, 53, 54}

Yoo et al.⁵⁵ analyze the overhead in the execution of Intel's STM.^{56,57} They identify four major sources of overhead: over-instrumentation, false sharing, amortization costs, and privatization-safety costs. False sharing, privatization-safety, and over-instrumentation are implementation artifacts that can be eliminated by using either finer-granularity bookkeeping, more refined analysis, or user annotations. Amortization costs are inherent overheads in STM that, as we demonstrated here, are not likely to be eliminated.

A large amount of research effort has been spent in analyzing the operations in TM systems. Recent software optimizations have managed to accelerate STM

possibly the single most powerful lever toward further reduction of STM overheads. Given the difficulty of similar problems explored by the research community such as alias analysis, escape analysis, and so on, this may be an uphill battle. Because the argument for TM hinges upon its simplicity and productivity benefits, we are deeply skeptical of any proposed solutions to performance problems that require extra work by the programmer.

We observed that the TM programming model itself, whether implemented in hardware or software, introduces complexities that limit the expected productivity gains, thus reducing the current incentive for migration to transactional programming and the justification at present for anything more than a small amount of hardware support. **Q**

ACKNOWLEDGMENTS

We would like to thank Pratap Pattnaik for his continuous support, Christoph von Praun for numerous discussions and work on benchmarks and runtimes, and Rajesh Bordawekar for the *b+tree* code implementation.

software transactional memory

→ why is it only a research toy?

REFERENCES

1. Herlihy, M., Moss, J.E.B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (May): 289–300.
2. Larus, J.R., Rajwar, R. 2006. *Transactional Memory*. Morgan Claypool.
3. Dice, D., Shalev, O., Shavit, N. 2006. Transactional Locking II. *DISC* (September): 194–208.
4. Harris, T., Fraser, K. 2003. Language support for lightweight transactions. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications* (October): 388–402.
5. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing* (July): 92–101.
6. Intel C++ STM compiler, prototype edition 2.0.; <http://softwarecommunity.intel.com/articles/eng/1460.htm/> (2008).
7. Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W.N., Scott, M.L. 2006. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester (March). Condensed version submitted for publication.
8. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B. 2006. Mcrt-stm: A high performance software transactional memory system for a multicore runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming* (March): 187–197.
9. Shavit, N., Touitou, D. 1995. Software Transactional Memory. In *Proceedings of the ACM Symposium of Principles of Distributed Computing*: 204–213.
10. Blundell, C., Devietti, J., Lewis, E.L., Martin, M.M.K. 2007. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*: 23–34.
11. Bobba, J., Goyal, N., Hill, M.D., Swift, M.M., Wood, D.A. 2008. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*: 127–138.
12. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (June): 102.
13. See reference 1.
14. Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K. 2007. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*: 69–80.
15. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture* (February).
16. Yen, L., Bobba, J., Marty, M.M., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture* (February).
17. Ceze, L., Tuck, J., Cascaval, C., Torrellas, J. 2006. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*: 237–238.
18. Tremblay, M., Chaudhry, S. 2008. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT. In *Proceedings of the IEEE International Solid-State Circuits Conference* (February).
19. Baugh, L., Neelakantam, N., Zilles, C. 2008. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*: 115–126.
20. Damron, P., Federova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D. 2006. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (October).

21. Saha, B., Adl-Tabatabai, A.R., Jacobson, Q. 2006. Architectural support for software transactional memory. In *Proceedings of the 39th Annual International Symposium on Microarchitecture* (December): 185-196.
22. Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., Scott, M.L. 2007. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*: 104-115.
23. STAMP benchmark suite. 2007. <http://stamp.stanford.edu/>.
24. The Lonestar benchmark suite. 2008. <http://iss.ices.utexas.edu/lonestar/>.
25. (IBM) XL C/C++ for Transactional Memory for AIX. 2008. www.alphaworks.ibm.com/tech/xlcstm/.
26. See reference 6.
27. See reference 3.
28. Blundell, C., Lewis, C., and Martin, M.M.K. 2006. Subtleties of transactional memory atomicity semantics. *IEEE TCCA Computer Architecture Letters* 5(2).
29. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, P.L. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the PLDI*: 211-222.
30. See reference 14.
31. See reference 23.
32. See reference 25.
33. Wu, P., Michael, M.M., von Praun, C., Nakaike, T., Bordawekar, R., Cain, H.W., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M.F., Wang, H.Y., Wang, K. 2008. Compiler and runtime techniques for software transactional memory optimization. To appear in *Concurrency and Computation: Practice and Experience*.
34. See reference 6.
35. See reference 3.
36. Yoo, R.M., Ni, Y., Welc, A., Saha, B., Adl-Tabatabai, A.R., Lee, H.-H.S. 2008. Kicking the tires of software transactional memory: why the going gets tough. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*.
37. See reference 36.
38. See reference 25.
39. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R., Moore, K.F., Saha, B. 2007. Enforcing isolation and ordering in STM. In *Proceedings of the Programming Language Design and Implementation Conference*: 78-88.
40. Harris, T., Plesko, M., Shinnar, A., Tarditi, D. 2003. Optimizing memory transactions. In *Proceedings of the Programming Language Design and Implementation Conference*: 388-402.
41. See reference 39.
42. See reference 40.
43. Zhang, R., Budimlić, Z., Scherer III, W.N. 2008. Commit phase in timestamp-based STM. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*: 326-335.
44. Shavit, N., Touitou, D. 1995. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*: 204-213.
45. See reference 3.
46. See reference 5.
47. See reference 4.
48. See reference 7.
49. Riegel, T., Fetzer, C., Felber, P. 2007. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures* (June).
50. Olszewski, M., Cutler, J., Steffan, J.G. 2007. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*: 365-375.
51. Spears, M.T., Michael, M.M., von Praun, C. 2008. RingSTM: Scalable transactions with a single atomic instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*: 275-284.
52. Felber, P., Fetzer, C., Mueller, U., Riegel, T., Suesskraut, M., Sturzrehm, H. 2007. Transactifying applications using an open compiler framework. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing* (August).
53. See reference 50.
54. Wang, C., Chein, W.-Y., Wu, Y., Saha, B., Adl-Tabatabai, A.R. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of International Symposium on Code Generation and Optimization*: 34-48.
55. See reference 36.
56. See reference 6.
57. See reference 8.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

CĂLIN CAȘCAVAL is a research staff member and manager of the Programming Models and Tools for Scalable Systems at IBM's T.J. Watson Research Center, where he has worked on several large-scale parallel computer projects. He

software transactional memory

→ why is it only a research toy?

developed the chip simulator for the Cyclops processor and worked on system software for the Blue Gene/C machine. Since 2003, he has been leading the Compilers team in the PERCS project (IBM's entry in the DARPA HPCS program). As part of this effort he is leading the development of the IBM xLUPC compiler and the Continuous Program Optimization project. He is also the project lead for transactional memory evaluation in IBM Research. He obtained an M.S. in computer engineering from Technical University Cluj-Napoca, Romania, in 1991, M.S. in computer science from West Virginia University in 1995, and Ph.D. from the University of Illinois at Urbana-Champaign in 2000.

COLIN BLUNDELL is a fifth-year graduate student at the University of Pennsylvania, where he is pursuing his Ph.D. under the supervision of Professor Milo Martin. His primary research interests are in multiprocessor performance and programmability, including memory consistency, cache coherence protocols, and hardware mechanisms for enabling optimistic concurrency.

HAROLD "TREY" CAIN is a research staff member at IBM's T.J. Watson Research Center, where he conducts research on the microarchitecture and memory system of highly multithreaded processors. He has coauthored more than 20 publications in the areas of multiprocessor memory system design, processor microarchitecture, simulation methodology, and the characterization of commercial server applications. Cain holds a Ph.D. and M.S. in computer science from the University of Wisconsin, and a B.S. from the College of William and Mary. His graduate research was selected as a 2004 "Top Pick" in computer architecture by *IEEE Micro*. His accent comes from the hills of eastern Kentucky.

MAGED M. MICHAEL is a research staff member at IBM's T.J. Watson Research Center. He received a Ph.D. in computer science from the University of Rochester. His research interests are primarily in concurrent algorithms, concurrent programming, and concurrent memory management. He is

the designer of well-known concurrent algorithms, including lock-free malloc, hazard pointers, and nonblocking algorithms for common data structures. His algorithms are used in commercial standard libraries, runtime systems, middleware, and realtime systems.

PENG WU is a research staff member at IBM's T.J. Watson Research Center and a member of the programming models and tools for scalable systems group. Her work has been centered around building a high-performance and high-productivity programming environment. Her past work on simdization has resulted in the first product release of an XL compiler with simdization capability and more than a dozen patents. More recently, she has been working on compiler and language supports for transactional memory and thread-level speculation. She received a Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 2001.

STEFANIE CHIRAS is a manager in IBM's Systems and Technology Group, leading a team responsible for technology and mainframe test. Prior to this role she was a research staff member in IBM Research, managing a team developing next-generation memory systems and incorporating emerging memory technologies. She joined IBM Research in 2001 as part of the Back End of Line Reliability team, after a postdoctoral position at Princeton University in the Princeton Materials Institute. She holds an M.S. and Ph.D. in materials science from the University of California at Santa Barbara.

SIDDHARTHA CHATTERJEE is director of the Austin Research Laboratory, one of IBM's eight worldwide research laboratories. He also serves as the research area strategist for systems architecture. He has held technical, managerial, executive, strategy, and staff positions during his time in the IBM Research Division. Most recently, he was senior manager of the Systems Solutions and Architecture group at IBM's T.J. Watson Research Center. Earlier, he was the leader of the Blue Gene performance team. Chatterjee received a B.Tech. in electronics and electrical communications engineering in 1985 from the Indian Institute of Technology, Kharagpur, and Ph.D. in computer science in 1991 from Carnegie Mellon University. Before joining IBM Research, he was a visiting scientist at the Research Institute for Advanced Computer Science (RIACS) in Mountain View, California, from 1991 through 1994, and was assistant and associate professor of computer science at the University of North Carolina at Chapel Hill from 1994 through 2001. He is currently an adjunct professor of computer science at the University of Texas at Austin.

© 2008 ACM 1542-7730/08/0900 \$5.00