

The promise of STM may likely be undermined by its overheads and workload applicabilities.

BY CĂLIN CAȘCAVAL, COLIN BLUNDELL, MAGED MICHAEL, HAROLD W. CAIN, PENG WU, STEFANIE CHIRAS, AND SIDDHARTHA CHATTERJEE

Software Transactional Memory: Why is it Only a Research Toy?

TRANSACTIONAL MEMORY (TM)¹³ 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-grained 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 the overall performance of TM is much 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 Rajwar¹⁶ present an overview of design trade-offs for implementations of transactional memory systems. We summarize some of the design choices here:

- ▶ Software-only (STM)^{7, 10, 12, 14, 18, 23, 25} is the focus here. While offering flexibility and no hardware cost, it leads to overhead in excess of most users' tolerance.

- ▶ Hardware-only (HTM)^{2, 4, 9, 13, 19, 20, 35} suffers from two major impediments: high implementation and verification costs lead to design risks too large to justify on a niche programming model; hardware capacity constraints lead to significant performance degradation when overflow occurs, and proposals for managing overflows (for example, signatures⁵) 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.^a

- ▶ Hybrid^{1, 6, 24, 28} is 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 systems are hardware-accelerated STMs. In this scenario, the transactional semantics are provided by the STM, and hardware primitives are only used to speed up critical performance bottlenecks in the STM. Such systems could offer an attractive solution if the cost of hardware primitives is modest and may be further amortized by other uses in the system.

Independent of these implementa-

^a Reuse of hardware for other purposes can also justify its inclusion, as the case may be for Sun's implementation of Scout Threading in the Rock processor.³²

Figure 1: STM operations.

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

(a) Pseudo-code for STM begin

```
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
```

(b) Pseudo-code for STM validate

```
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
```

(c) Pseudo-code for STM read barrier

```
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
```

(d) Pseudo-code for STM end

tion 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 non-transactional 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, due to 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 exclud-

ing private data. Furthermore, the non-determinism 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 our view that 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 STAMP³⁰ and Lonestar¹⁷ 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

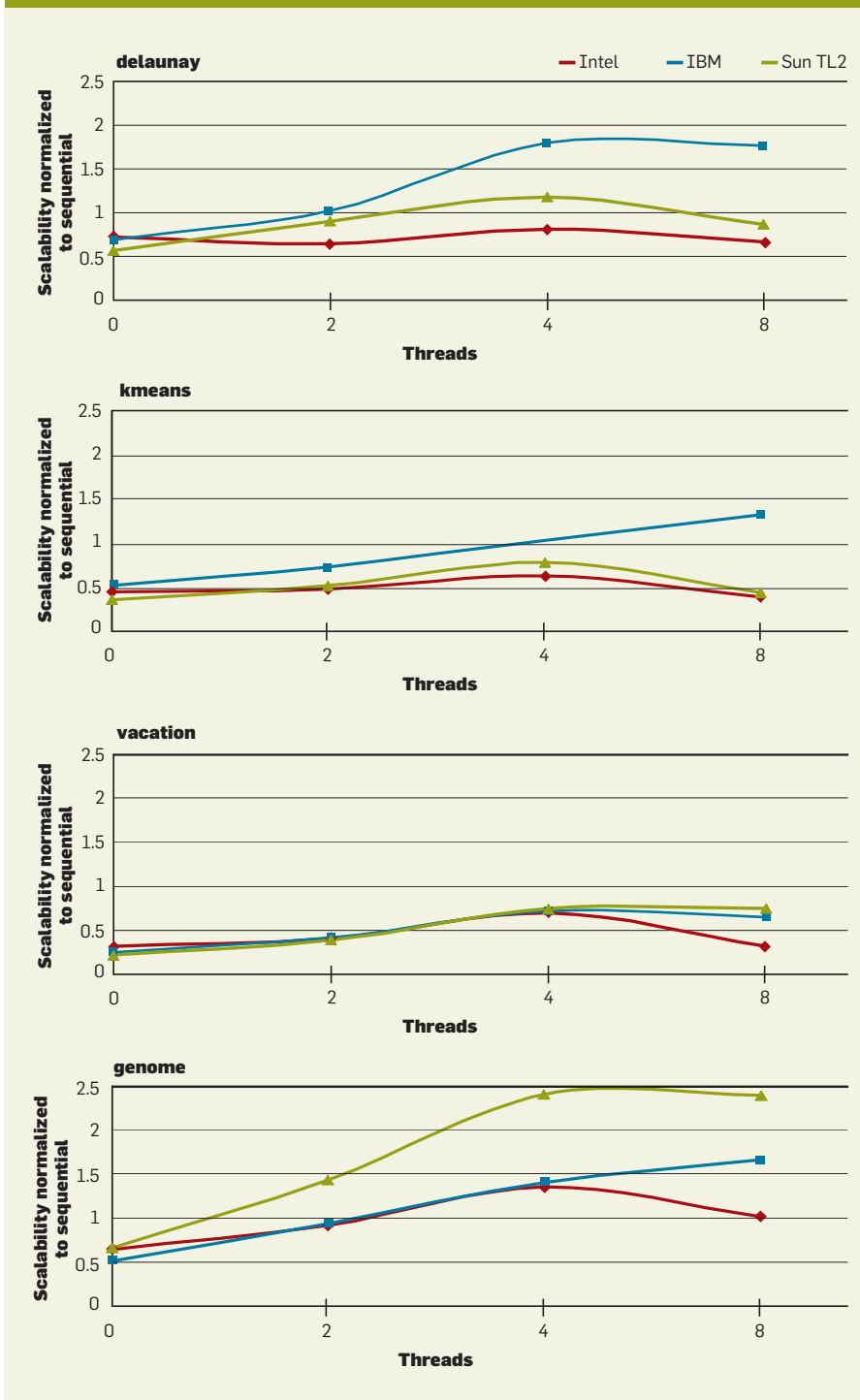
state of the art STM runtime system and compiler framework, the freely available IBM STM.³¹ Here, we describe this experience, starting with a discussion of STM algorithms and design decisions. We then compare the performance of this STM with two other state of the art implementations (the Intel STM¹⁴ and the Sun TL2 STM⁷) 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 pseudo-code for the main operations executed by a typical STM is illustrated in Figure 1. We show 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 an STM for system programmers is that it offers flexibility in implementing different mechanisms and policies for these operations. For

Figure 2: Scalability results for three STM runtimes on a quad-core Intel Xeon server: IBM, Intel STM v2, and Sun TL2.



end users, the advantage of an STM is that it offers an environment to transactionalize (that is, porting to TM) their applications without incurring extra hardware cost or waiting for such hardware to be developed.

Conversely, an 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 1c). 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:** In order to avoid incurring high STM overheads, non-transactional accesses (such as 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:

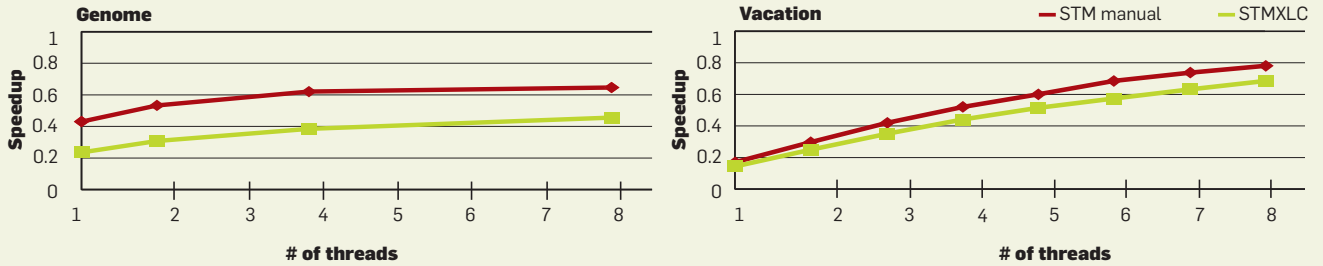
► **Weak atomicity:** Typically the STM runtime libraries cannot detect conflicts between transactions and non-transactional accesses. Thus, the semantics of atomicity are weakened to allow undetected conflicts with non-transactional accesses (referred to as weak atomicity³), or equivalently put the 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 privately—or non-transactionally in general, by using locks. For some STM designs, once a location is accessed transactionally, it must continue to be accessed transactionally. With some STM designs, 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 from other locations.

► **Legacy binaries:** STM needs to observe all memory activities of the transactional regions to ensure atomicity and isolation. STMs that achieve this observation by code instrumentation gener-

Figure 3: Scalability results for manual and compiler instrumented benchmarks on AIX PowerPC with IBM XLCTM compiler.



ally 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

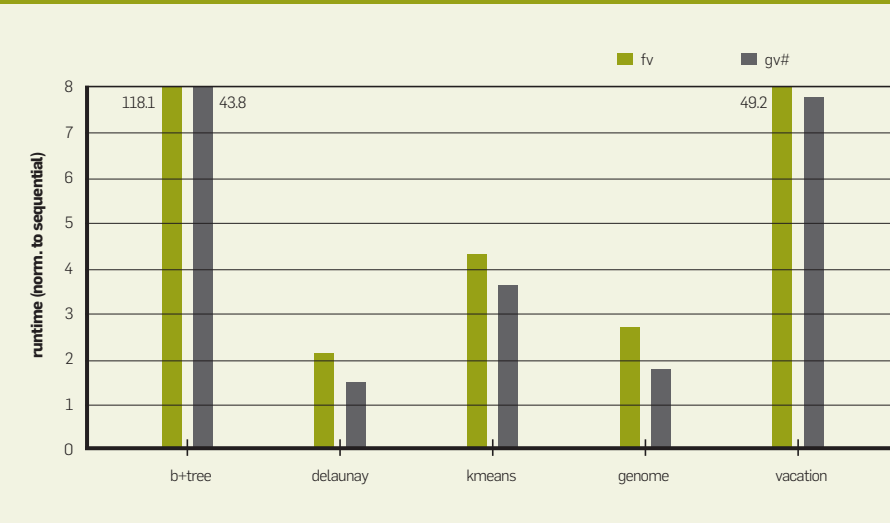
Here we use the following set of benchmarks:

- *b+tree* is an implementation of database indexing operations on a b-tree data structure for which the data is stored only on the tree leaves. This implementation uses coarse-grain transactions for every tree operation. Each b+ tree operation starts from the tree root and descends down 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.¹⁵ 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 suite¹⁹

Figure 4: Single-threaded overhead of the STM algorithms.



version 0.9.4. For a detailed description of these benchmarks see STAMP.³⁰

Baseline Performance. In Figure 2 we present a performance comparison of three STMs: the IBM,^{31, 34} Intel,¹⁴ and Sun's TL2⁷ STMs. The runs are on a quad-core, two-way hyperthreaded Intel Xeon 2.3GHz box running Linux Fedora Core 6. In these runs, we used the manually instrumented versions of the codes that 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, the curves for the Intel STM are from codes instrumented by its compiler, which incur additional barrier overheads due to compiler instrumentation.³⁶ The graphs are scalability curves with respect to the serial, non-transactionalized 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*. However, the overall performance obtained is very low: on *kmeans* the IBM

STM barely attains single thread performance at 4 threads, while on *vacation* none of the STMs actually overcome the overhead of transactional memory even with 8 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 due to conservativeness of compiler analysis, as also observed in Yoo.³⁶

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.³¹

Figure 5: Percentage of time spent in different STM operations.

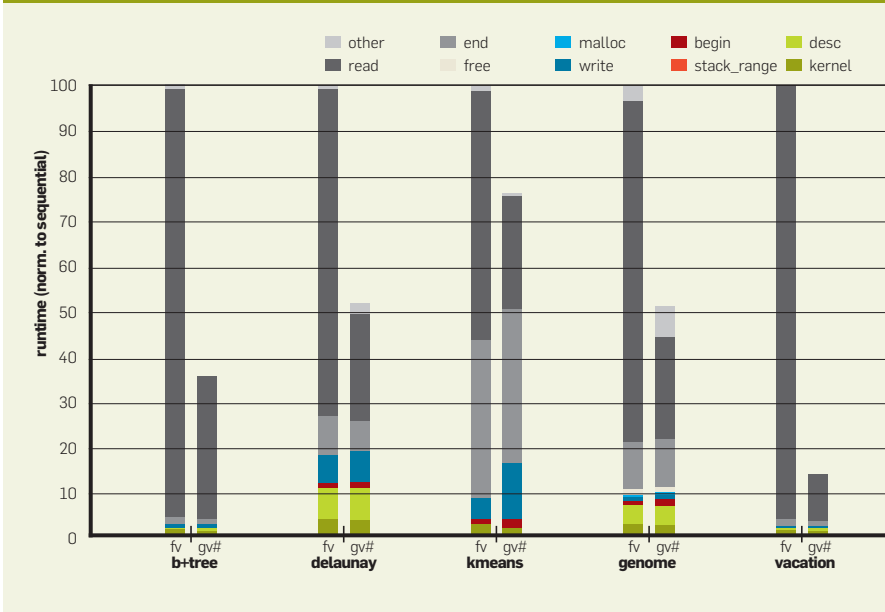
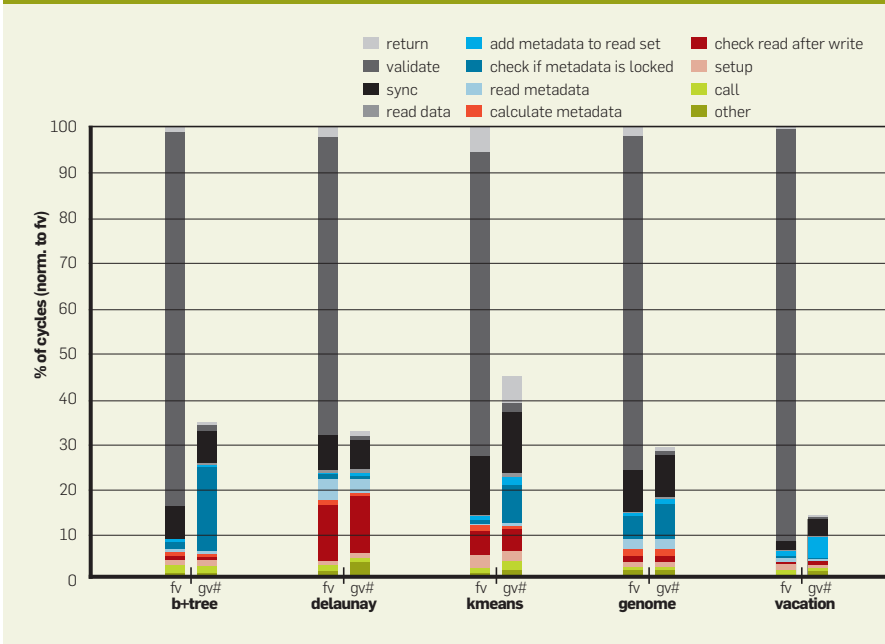


Figure 6: Percentage of time spent in STM read sub-operations.



Compiler over-instrumentation is more pronounced in traditional, unmanaged languages, such as C and C++, where a compiler instrumentation without interprocedural analysis may end up instrumenting every memory reference in the transactional region (except for stack accesses). Indeed, 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 instruction level and eliminate any other non-determinism 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 trade-offs 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 due to 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 6 gives a fine-grain 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 sync 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 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 that 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 which are complementary to STM hardware acceleration.

► *Redundant barrier elimination.* One technique is to eliminate barriers to thread-local objects through escape analysis. Such analysis is typically quite effective 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 intra-procedural escape analysis can eliminate 40–50% of write barriers in *vacation*, *genome*, and *b+tree*. However, its impact on performance is more limited: from negligible to 12%. To target redundant read-barriers, a whole-program analysis called Not-Accessed-In-Transaction analysis²⁷ 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^{11,27} and runtime filtering of stack references and duplicate references;¹¹

► *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% overall improvement across our benchmark suite;

► *Commit sequence optimizations.* Eliminating unnecessary global version number updates³⁷ improves the overall performance of several micro-benchmarks by up to 14%.

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

Related Work

The first STM system was proposed by Shavit and Touitou²⁶ and is based on object ownership. The protocol is static, which is a significant shortcoming that has been overcome by subsequently pro-

posed STM systems.⁷ Conflict detection is simplified significantly by the static nature because conflicts can be ruled out already when ownership records are acquired (at transaction start).

DSTM¹² 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 Fraser¹⁰), 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 RSTM¹⁸ 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 co-locates it with the non-transactional 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

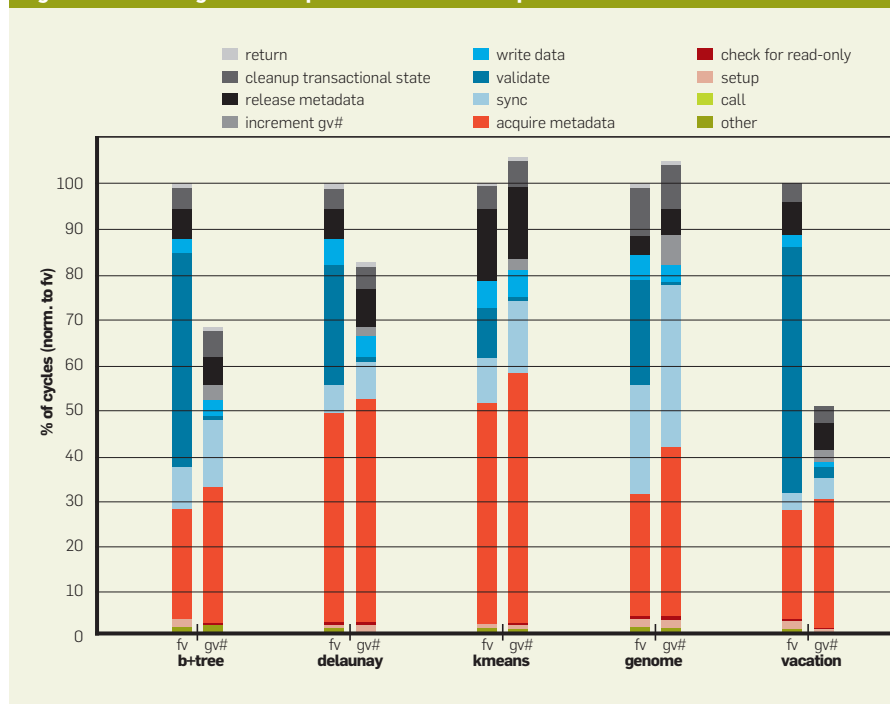
hence this scheme is amenable for use in environments where memory management is explicit.

Recent work explored algorithmic optimizations and/or alternative implementations of the basic STM algorithms described here. Riegel et al. propose the use of real-time clocks to enhance the STM scalability using a global version number.²² JudoSTM²¹ and RingSTM²⁹ 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 due to imprecise conflict detection. Several proposals have been made for STMs that operate via dynamic binary rewriting in order to allow the usage of STM on legacy binaries.^{8,21,33}

Yoo et. al³⁶ analyze the overhead in the execution of Intel’s STM.^{14,23} 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 either using finer granularity bookkeeping, more refined analysis, or user annotations. Amortization costs are inherent overheads in an STM that, as we demonstrated here, are not likely to be eliminated.

A large amount of research effort has been spent in analyzing the opera-

Figure 7: Percentage of time spent in STM end sub-operations.




tions in TM systems. Recent software optimizations have managed to accelerate STM performance by 2%–15%. 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 overheads 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—possibly the single most powerful lever toward further reduction of STM overheads. However, 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. And 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.

Acknowledgments

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

References

- Baugh, L., Neelakantam, N., and Zilles, C. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, 2008, 115–126.
- Blundell, C., Devietti, J., Lewis, E.L., Martin, M.M.K. 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*. ACM, NY, 2007.
- Blundell, C., Lewis, C., and Martin, M.M.K. Subtleties of transactional memory atomicity semantics. *IEEE TCCA Computer Architecture Letters* 5, 2 (Nov 2006).
- Bobba, J., Goyal, N., Hill, M.D., Swift, M.M., and Wood, D.A. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th International Symposium on Computer Architecture*. IEEE Computer Society, Washington, D.C., 2008, 127–138.
- Ceze, L., Tuck, J., Cascaval, C., Torrellas, J. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, NY, 2006, 237–238.
- Damron, P., Federova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.
- Dice, D., Shalev, O., and Shavit, N. Transactional Locking II. *DISC*, Sept. 2006, 194–208.
- Felber, P., Fetzer, C., Mueller, U., Riegel, T., Suesskraut, M., and Sturzhelm, H. Transactifying applications using an open compiler framework. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing*. Aug. 2007.
- Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, June 2004, 102.
- Harris, T. and Fraser, K. Language support for lightweight transactions. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications*. Oct. 2003, 388–402.
- Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *Proceedings of the Programming Language Design and Implementation Conference*. 2003, 388–402.
- 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*. July 2003, 92–101.
- Herlihy, M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. May 1993.
- Intel C++ STM compiler, prototype edition 2.0.; <http://softwarecommunity.intel.com/articles/eng/1460.htm/> (2008).
- Kulkarni, M., Pingali, K., Walter, B., Ramnarayanan, G., Bala, K., and Chew, P.L. Optimistic parallelism requires abstractions. In *Proceedings of the PLDI 2007*. ACM, NY, 2007, 211–222.
- Larus, J.R., and Rajwar, R. *Transactional Memory*. Morgan Claypool, 2006.
- The Lonestar benchmark suite; <http://iss.ices.utexas.edu/lonestar/> (2008).
- Marathe, V.J., Spear, M.F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W.N., and Scott, M.L. Lowering the overhead of software transactional memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006. Condensed version submitted for publication.
- Minh, C.C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, NY, 2007, 69–80.
- Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., and Wood, D.A. LogTM: Log-based transactional memory. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, Feb 2006.
- Olszewski, M., Cutler, J., Steffan, J.G. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. 2007. IEEE Computer Society, Washington D.C., 365–375.
- Riegel, T., Fetzer, C., and Felber, P. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.
- Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., and Hertzberg, B. Mrcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*. Mar. 2006, ACM, NY, 187–197.
- Saha, B., Adl-Tabatabai, A.R., and Jacobson, Q. Architectural support for software transactional memory. In *Proceedings of the 39th Annual International Symposium on Microarchitecture*. Dec. 2006, 185–196.
- Shavit, N., and Touitou, D. Software Transactional Memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 1995.
- Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. ACM, NY, 1995.
- Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R., Moore, K.F., and Saha, B. Enforcing isolation and ordering in STM. In *Proceedings of Proceedings of the Programming Language Design and Implementation Conference*. ACM, 2007, 78–88.
- Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarakadas, S., and Scott, M.L. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, NY, 2007, 104–115.
- Spears, M.T., Michael, M.M., and von Praun, C. Ringstm: Scalable transactions with a single atomic instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, NY, 275–284.
- STAMP benchmark; <http://stamp.stanford.edu/> (2007).
- (IBM) XL C/C++ for Transactional Memory for AIX; <http://www.alphaworks.ibm.com/tech/xlccstm/> (2008).
- Tremblay, M. and Chaudhry, S. A third generation 65nm 16-core 32-thread plus 32-scout-thread CMT. In *Proceedings of the IEEE International Solid-State Circuits Conference*. Feb. 2008.
- Wang, C., Chein, W.-Y., Wu, Y., Saha, B., and Adl-Tabatabai, A.R. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of International Symposium on Code Generation and Optimization*. 2007, 34–48.
- Wu, P., Michael, M.M., von Praun, C., Nakaïke, T., Bordawekar, R., Cain, H.W., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M.F., Wang, H.Y., and Wang, K. Compiler and runtime techniques for software transactional memory optimization. To appear in *Concurrency and Computation: Practice and Experience*, 2008.
- Yen, L., Bobba, J., Marty, M.M., Moore, K.E., Volos, H., Hill, M.D., Swift, M.M., and Wood, D.A. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*. Feb 2007.
- 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. *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2008.
- Zhang, R., Budimlic, Z. and Scherer III, W.N. Commit phase in timestamp-based STM. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, NY, 326–335.

Çâlin Caşcaval (cascaval@us.ibm.com) is a Research Staff Member and Manager of Programming Models and Tools for Scalable Systems at IBM TJ Watson Research Center, Yorktown Heights, NY.

Colin Blundell is a member of the Architecture and Compilers Group, Department of Computer and Information Science, University of Pennsylvania.

Maged Michael is a Research Staff Research Member at IBM TJ Watson Research Center, Yorktown Heights, NY.

Trey Cain is a Research Staff Member at IBM TJ Watson Research Center, Yorktown Heights, NY.

Peng Wu is a Research Staff Member at IBM TJ Watson Research Center, Yorktown Heights, NY.

Stefanie Chiras is a manager in IBM's Systems and Technology Group.

Siddhartha Chatterjee is director of the Austin Research Laboratory, IBM Research, Austin, TX.