Transactional Tools for the Third Decade *

Matthew Kilgore, Stephen Louie, Chao Wang, Tingzhe Zhou, Wenjia Ruan, Yujie Liu, and Michael Spear
Lehigh University
{mak209, srl214, chw412, tiz214, wer210, yul510, spear}@cse.lehigh.edu

Abstract

In this paper, we present the current state of a variety of software tools that we are making available to the broad research community. Our intent is to ensure that researchers in Transactional Memory (TM) and related fields have a common baseline that is both easy to use and appropriate for implementing new algorithms and testing hypotheses.

The most significant contribution is a transactionalized C++ Standard Template Library. We also provide a proper and extensible lazy software TM implementation, a common build environment, a repackaging of several benchmarks, and a transactional thread-level speculation infrastructure. In total, we believe this creates a suitable baseline for researchers in this “third decade” of Transactional Memory.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Transactional Memory, Synchronization, GCC, C++, Standard Template Library

1. Introduction

The term “Transactional Memory” (TM) was first coined in 1993 [11]. Researchers have spent the subsequent years exploring first what TM ought to be, then how to make it fast. With the subsequent arrival of both hardware [13] and compiler [9] support for TM in commodity products, the third decade of TM research appears to be one of how to use TM most effectively. This does not obviate continued research into hardware and software TM implementations, but it does change how research is conducted: any new proposal ought to be immediately compatible with existing tools, or else it must justify its deviation from the standard.

Surprisingly, the existing TM tools do not make this easy. Extending the GCC TM to support lazy writes requires significant nuance in data structure design. Transactionalizing real-world applications is similarly nontrivial. Even standard benchmarks are not necessarily easy to use. We worry that the difficulty of the tools results in a slower pace of research, if not bad research.

Our goal is to provide a baseline that remedies these problems. The most significant contribution of this work is the release of a TM-compatible C++ Standard Template Library (STL). While the work to make the STL transaction-safe is relatively straightforward, the corner cases in which transactions misbehave are nontrivial, and merit discussion. To motivate the need for a transactional STL, we show that using the STL in place of traditional ad-hoc TM-friendly collections can have surprising performance consequences. If real-world developers use the STL, and TM researchers do not, our conclusions are likely to diverge.

Our second contribution is a researcher-friendly variant of the GCC TM implementation. Our implementation includes a proper lazy software TM implementation, and is organized in a manner that is more amenable to researchers. We share some cautionary tales about the creation of this implementation, which we hope will convince fellow researchers of the importance of a good baseline implementation.

Our third contribution is a novel framework for thread-level speculation using TM. The benefit of this framework is twofold: first, it is open-source and built on standard technologies, which makes it generalizable. Second, it shows how a researcher-friendly GCC TM implementation makes it possible to re-purpose TM mechanisms to related domains.

Finally, we provide a pre-packaged set of TM benchmarks (STAMP [18], PARSEC [2], and Memcached [22]) in a single place, with a unified build system. This provides an easy way for researchers to be sure they are running the same tests, in a reproducible fashion.

2. Transactionalizing the STL

The Draft C++ TM Specification [1] defines how lexically scoped transactions are marked by the programmer. The specification makes no assumptions about how TM is implemented. Thus it must assume that a transactional program may run on a system without hardware support, in which case per-memory-access instrumentation is needed.

The challenge this raises is that certain operations cannot be undone. These *unsafe* operations include I/O and inter-thread communication via `atomic` variables. Without hardware support, these operations cannot be detected at runtime. To support software TM, the compiler must statically ensure that unsafe operations are not issued from a transaction. When a transaction calls functions that are defined in other modules, this static checking requires additional language machinery. To that end, the specification introduces an annotation on functions, `transaction_safe`, which instructs the compiler to (a) verify the absence of operations that cannot be undone, and (b) produce a version of the function in which all shared memory accesses are instrumented.[1]

A final wrinkle is that in templated code, the transaction-safety of a function can depend on how the code is instantiated. As a simple example, consider the following template:

```
template <class T>
class Foo<T> {
  T t;
  int operation() { return t.getData(); }
}
```

In this code, `Foo::operation()` is transaction-safe only if `T::getData()` is transaction safe. If the programmer intends

* All software artifacts related to this work can be found at `https://github.com/mfs409/transmem`

[1] The upcoming revision to the draft specification will make these annotations part of the function's type.

for `Foo` to be instantiated with many different classes `T`, of which some do not have a safe `getData` function, then it is not correct to annotate `operation()` as being transaction-safe. The specification solves this through limited inference: within a translation unit (e.g., a `.cc` file and all of its included headers), the safety of functions is inferred by the compiler. Annotations are only required for functions that are declared within a translation unit, but defined elsewhere (e.g., separate compilation).

2.1 A TM-Friendly STL

To identify the modifications needed to make the STL work with TM, we developed sequential programs to instantiate every method of each of the following containers: deque, list, map, pair, tuple, unordered_map, unordered_multiset, unordered_set, and vector. We then transformed these sequential programs, each consisting of roughly 1500 lines of code, into parallel programs in which all code ran within transactions. Compilation errors for this second version of the code identified the modifications needed to make each STL container transaction-safe.

We tested every method of each container, as defined in the upcoming C++14 specification. Depending on the container, this could be as high as 70 distinct methods. We determined that 18 modifications were required:

- The current specification does not define `std::abort()` to be transaction-safe. When using `std::list`, a program should terminate immediately if an operation is performed on two lists that do not use the same allocator. Since the next draft of the C++ TM specification will declare `std::abort` to be safe, we modified it accordingly.
- `std::list` uses an auxiliary class to define its underlying list, and that underlying class defines several functions outside of the template header file. Five `transaction_safe` annotations were required. Another five were needed in the red-black tree used by `std::set`, and another two annotations in the hashtable code used by `std::unordered_set`, for the same reason.
- GCC does not yet provide a transaction-safe `memcmp()`. We added a safe `memcmp()`, which enabled `std::equal` to be used safely within STL containers.
- GCC's STL implementation uses helper functions to normalize the throwing of exceptions across containers. We annotated 4 functions for reasons similar to "out-lined code".

In total, this effort introduced less than 50 new lines of code, including comments. We also identified a few (surmountable) problems that will be instructive to developers:

- The specification insists that transactional and nontransactional versions of a function be generated from the same source. This can introduce performance penalties. For example, in string functions, nontransactional code might wish to use SIMD instructions that are not compatible with TM. Ni et al. proposed a `transaction_wrap` mechanism, through which different (but functionally equivalent) code bodies can be provided for the transactional and nontransactional versions of a function [19]. This feature was beneficial to `std::equal`. Coupled with a prior study on transactionalizing memcached [22], we believe there is sufficient justification to add this feature to the specification.
- While STL containers are the preferred foundation upon which to build C++ code, they are not always designed with scalability in mind.[2] In one surprising example, the C++11 standard introduced the requirement for `std::list` to provide an $O(1)$ size function. This, in turn, necessitates a per-list counter, modified on every insert and removal. Similar bottlenecks exist in more scalable data structures, such as the red-black tree (`std::set`) and hashtable (`std::unordered_set`).

[2] A new C++ Study Group has begun working on this problem.

2.2 An Open Problem: Exceptions, Strings, and Memory Leaks

We also encountered one insurmountable problem, related to backwards compatibility. In December of 2014, the GCC implementation of `std::string` changed in a way that both (a) breaks backward compatibility, and (b) is appealing to TM. Prior to the change, `std::string` used reference counts, achieved via `atomic` variables. Consequently, it was not possible to make `std::string` methods transaction safe without introducing a penalty for nontransactional code (i.e., by using transactions within the `std::string` implementation, for every reference count operation).

This change to `std::string`, which was necessary for C++11 compliance, comes with a wrinkle: a program may use some modules compiled to the old ABI, and others compiled to the new. The linker will ensure that the symbols differ, so that new-ABI code, using the new `string`, is not passed a reference to an instance of the old and incompatible `string`. However, exceptions us `std::string` internally, and exceptions can be thrown from old-ABI code and caught in new-ABI code (or vice versa). The solution in GCC is to uniformly use old-ABI `string` objects within exception objects.

Unfortunately, this implies that exceptions cannot be created, thrown, caught, or re-thrown from transactions (otherwise, the transaction must be able to access a volatile reference count, which is not transaction-safe). The only workaround at the present time is to outline any STL code that deals with exceptions (for modularity, GCC already does this), and then mark that code as `transaction_pure`. Software transactions may call pure functions, because it is assumed that such functions are free of side effects.

This trick, however, also carries a cost: in a pure function, calls to `malloc` are not monitored by the TM, and hence, if the calling transaction aborts, the allocation will not be reclaimed; there can be memory leaks. This applies to our "pure" exception code: if an exception is thrown by transactional STL code, and the caller aborts before reclaiming the exception object, the memory of the exception object will be leaked. Of course, this problem only affects code that throws exceptions from transactions. The common-case behavior of the STL is without memory leaks. A potential solution is to employ onAbort handlers [3] within the STL code.

3. STAMP+STL

Armed with a transactional STL, we explored the opportunity to replace STAMP code with calls to C++ libraries. Two immediate opportunities were the use of `std::sort` in Bayes, and `std::mt19937` as the random number generator in all benchmarks. This saved 250 lines of code. Furthermore, since these objects are implemented in headers, we did not require any modifications to C++ library code when calling these functions from transactions. Previously, STAMP's `mt19937` code was treated as pure, resulting in nondeterministic behavior when a transaction aborted after producing a random number.

We then turned our attention to standard libraries. With the exception of KMeans and SSCA2, all STAMP benchmarks rely on custom data structures that have equivalents in the C++ STL.

Eliminating these objects reduced the code size of STAMP by 33%, to 13K lines of code. More details are provided below:

- Ordered lists: Bayes, Intruder, Vacation, and Yada used a sorted linked list to implement a set. The C++ STL does not offer a sorted list set abstraction (`std::list` is unsorted), but `std::set`, which uses a red-black tree as its underlying data structure, offers the same interface as STAMP's list. Genome uses a custom hash table with a fixed number of buckets, where each bucket held a linked list. Here, too, we replaced the lists with `std::set`, resulting in an array of lists. Lastly, in Labyrinth, a list of vectors of vectors was replaced with a `std::set` of `std::vector`s to `std::vector`s. This effort also encountered an invalid cast from `long` to `void*` in Bayes (STAMP lists hold `void*` elements).
- Maps: STAMP uses either AVL trees or red-black trees to implement maps. We replaced these with `std::map` in Vacation, Intruder, and Yada. There was no functional difference between the choice of AVL or red-black tree, and hence the use of `std::map`, which uses a red-black tree, was sufficient.
- Vectors: Genome, Labyrinth, Intruder, Bayes, and Yada used vectors, which we replaced with `std::vector`. STAMP's vector takes a capacity hint in its allocation function, which we achieved instead by using the `vector::reserve()` function.
- Hash tables: Genome used custom a custom hashtable, which we replaced with `std::unordered_set`. While the STAMP hashtable is a key/value store, Genome only requires a key store, hence we did not use `std::unordered_map`.
- Bitmaps: Genome and Bayes both use a bitmap object. Unfortunately, `std::bitmap` requires the bitmap size at compile time. Instead, we used `std::vector<bool>`.
- Pairs: Vacation, Intruder, Yada, and Labyrinth used a custom pair object, which we replaced with `std::pair`.
- Queues: The STAMP queue object used by Labyrinth, Intruder, Bayes, and Yada uses a fixed size array, which resizes upon insertion into a full queue. Instead, we used `std::queue`, which is an adapter class atop `std::deque`. There were a few API changes: STAMP's queue uses a single function, `pop`, where the STL uses the sequence `front(); pop();`; STAMP's queue offers a `clear()` method, whereas the `std::queue` requires a loop of `pop`s to achieve the same effect; and STAMP's `pop` returns NULL when issued on an empty queue, necessitating that all pops from `std::queue` first checked for an empty queue. Lastly, STAMP's queue has a `shuffle` method, for randomizing the order of elements. We achieved this by moving all data to a vector, shuffling the vector via the same algorithm as STAMP's queue, and then copying the vector back into the queue.
- Heaps: Yada uses a priority queue (min heap). We replaced this with `std::multiset`, to capture the requirement that the heap can have several elements with the same value.

In addition to eliminating code and making STAMP resemble real-world C++ programs, STL-STAMP offers one more benefit: the use of C++ containers instead of ad-hoc C data structures enables changing data structures in a constant number of source code edits. Previously, changing a data structure required changes to every use of the data structure. Using STL containers, it is only necessary to change the declaration and constructor call. This will facilitate research into new transactional data structures (e.g., variants of `set` that do not have a global counter).

4. A Better Lazy Software TM Implementation

In 2011, Kestor et al. presented a shim for translating calls by the Intel TM Compiler into calls to the RSTM library [14]. In our opinion, this work has been largely ignored, resulting in some erroneous research.

Kestor identified three challenges for Lazy TM, one of which was also reported by Yoo et al. [25]:

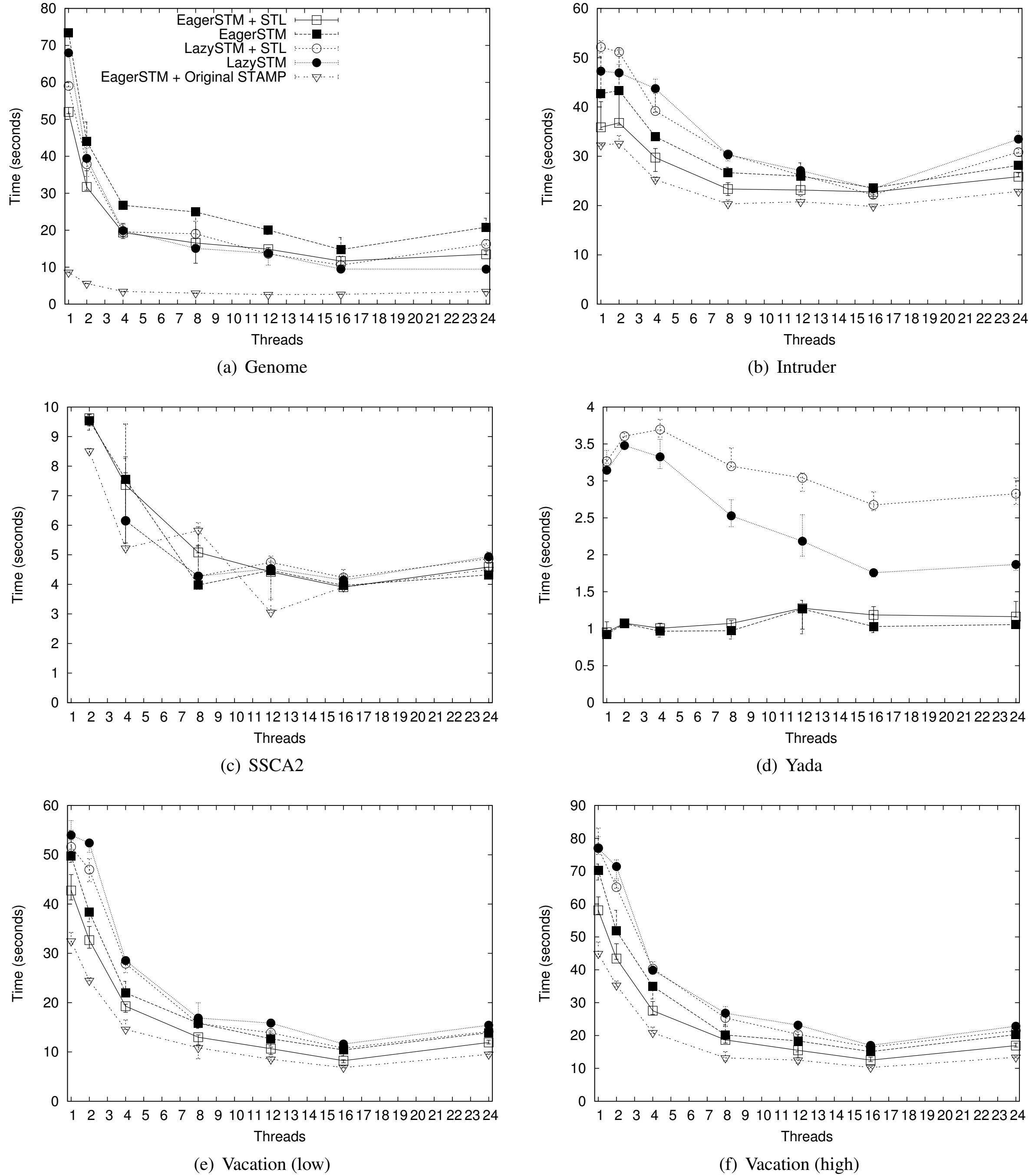
- Pointer aliasing of writes to the stack can violate correctness.
- Value validation of aliased stack reads can cause unnecessary aborts.
- Write-set implementations for Lazy software TM require more nuance.

Given that GCC-TM is open-source, we sought to create a lazy TM within the GCC infrastructure, rather than connecting GCC-TM to RSTM via a level of indirection.[3] Since we required both lazy orec-based TM [6, 8, 23] and NOrec [5], we began by creating a "researcher-friendly" GCC-TM library. Whereas GCC-TM must be compiled as part of a full GCC source checkout, our library is a standalone build, using the same GCC-TM source code, but a custom Makefile. Though a trivial task, this offers huge benefit, since researchers can independently manage multiple TM implementations, without requiring gigabytes of storage and lengthy configure/compilation times.

In regard to Kestor's first challenge, pointer aliasing, we developed test cases to show when this can occur. Suppose there exists a function `f`, which takes a single parameter of type `int*`, named `p`. Let `f` be called with `p` being the address of a heap variable: in this case, `f` must use instrumentation to read and write at the location referenced by `p`, or else concurrent transactional accesses to that location will be racy. However, if `f` is called from a transaction, but passes the address of a transaction-local value (i.e., a variable declared within the lexical scope of the transaction)) as `p`, then a problem arises: the compiler knows that `p` refers to a local, and hence subsequent reads of the local variable are not instrumented (i.e., are not transactional reads). To ensure the subsequent reads observe the updated value, it is necessary for transactional writes to check if a variable is transaction-local, and update it directly if so. A similar problem arises for non-shared stack variables that are not local to the transaction. These must be undo-logged, in case of rollback, but again must be updated immediately.

The second challenge is more subtle, in that it does not apply to all lazy TMs, only to NOrec. In NOrec, a transaction logs the locations it reads, and the values it observes. We found that our first (naive) NOrec implementation scaled poorly. We discovered that our benchmark performed a transactional read to a thread-local value on the transaction's stack. The relevant function subsequently returned, another function was called, and the address of the previously-read stack location changed. At this point, any validation of the transaction would cause an erroneous abort. However, aborts only happen in response to concurrent commits by other writer transactions. Thus single-threaded code runs without any slowdown, despite this bug. Worse, concurrent code does not hang: no transaction can successfully validate, but a writing transaction may commit if there were no successful commits between its start and commit. In effect, the behavior of the program reduced to sequential, with many transactions starting, one committing, the remainder validating and erroneously aborting, and then the process repeating. It was necessary for us to add a custom stack filtering step to transactional reads in our NOrec implementation. We hope that Hybrid TM researchers who are comparing to Hybrid NOrec [4] have not made this same mistake!

[3] Recall that this approach was not possible in 2011, when GCC-TM was not available.



(a) Genome

(b) Intruder

(c) SSCA2

(d) Yada

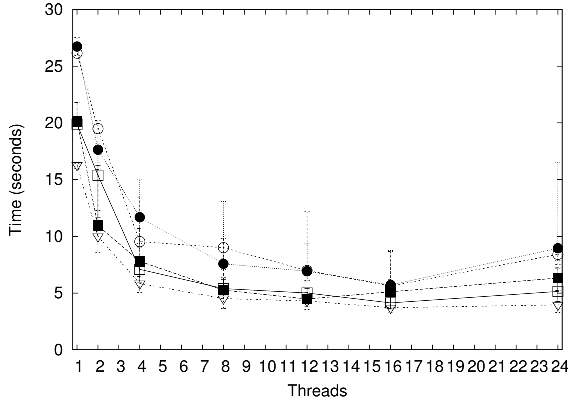
(e) Vacation (low)

(f) Vacation (high)

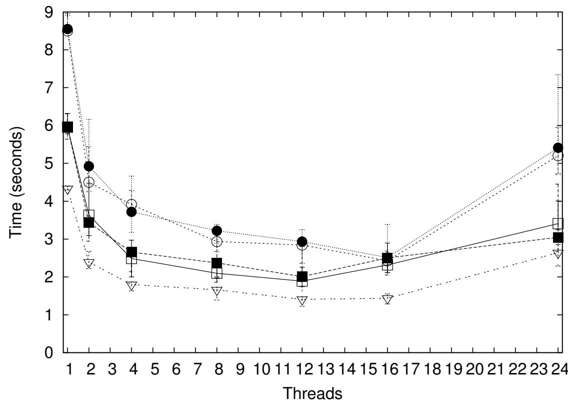
Figure 1: STAMP results on Westmere dual-chip system (1/2)

Lastly, the write set must be able to handle mixed-granularity accesses (e.g., writing a byte and then reading the enclosing word), without ever performing write-back of bytes that were not explicitly read by a transaction. Most library-based TM implementations incorrectly operate at the granularity of words, and most manually instrumented workloads do not make dangerous accesses. However, GCC can, and does, make mixed granularity accesses, in ways that are completely legitimate and necessary for correct program execution. To ensure correctness, we extended the RSTM hash table to store 64-byte blocks, along with a 64-bit mask to track which bytes within a block are valid. We performed a set of obvious sequential optimizations to this implementation.

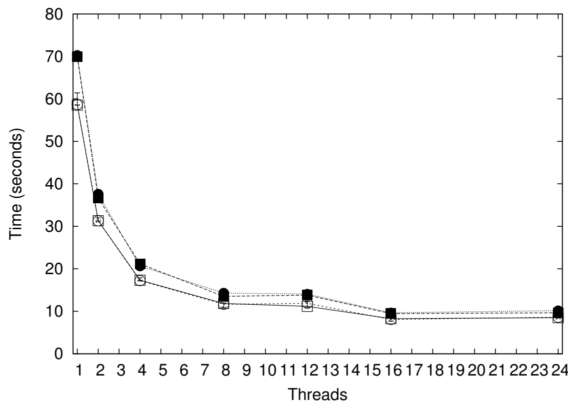
To summarize, our framework is source-code and binary compatible with GCC-TM, is easy to build, and correctly supports hardware TM (i.e., Intel TSX), eager software TM (e.g., the baseline GCC-TM or TLRW [7]), and lazy software TM (to include NOrec). We believe it is the best available starting point for new TM algorithm designers who wish to use GCC-TM.



(a) KMeans (low)



(b) KMeans (high)



(c) Labyrinth

Figure 2: STAMP results on Westmere dual-chip system (2/2)

5. Performance Evaluation

As stated earlier, we view the transactional STL as the most significant contribution in our new toolkit. Let us take a moment to analyze its performance.

5.1 Experimental Configuration

We consider two hardware systems. Experiments labeled "Westmere" were performed on a system with two Intel Xeon X5650 chips running at 2.67 GHz. This system has 12 cores/24 threads. Experiments labeled "Haswell" were performed on a system with a single Intel Core i7-4770 CPU running at 3.40 GHz. The "Haswell" machine supports hardware TM. Both machines ran Ubuntu Linux 13.04, Linux kernel version 3.8.0. All benchmarks were compiled with the latest pre-release GCC 5.0. Code was compiled with –O2 optimizations. All experiments reported in this paper are the average of five trials.

We compare up to three TM implementations on each system: "HTM", running only on the Haswell system, uses Intel TSX [13]. "EagerSTM" refers to the standard GCC software TM, a privatization-safe version of TinySTM [8]. "LazySTM" modifies GCC's software TM to use commit-time locking and redo logging, effectively providing a privatization-safe version of Patient TM [23].

We similarly compare three versions of STAMP. "+ Original" refers to STAMP 0.9.10, from 2008. Curves lacking a "+" used C++STAMP [21]. "+ STL" replaces C++STAMP data structures with STL containers. All benchmarks were run with recommended non-simulator parameters. For KMeans and Vacation, we used both the high- and low-contention settings. Note that "+ Original" Yada and Labyrinth are not shown, as they use `transaction_restart`, which is incompatible with the GCC's TM, and Bayes is not presented, since it is nondeterministic.

5.2 STAMP Performance

Figures 1, 2, 3, and 4 present STAMP performance on the Westmere and Haswell systems, respectively. Surprisingly, the use of the STL could either increase or decrease latency, depending on the workload. These outcomes both make sense: for contended data structures, the use of exact counters in the STL to report collection size can be a bottleneck. On the other hand, template code is more aggressively inlined, and hence there ought to be lower latency than in previous STAMP versions.

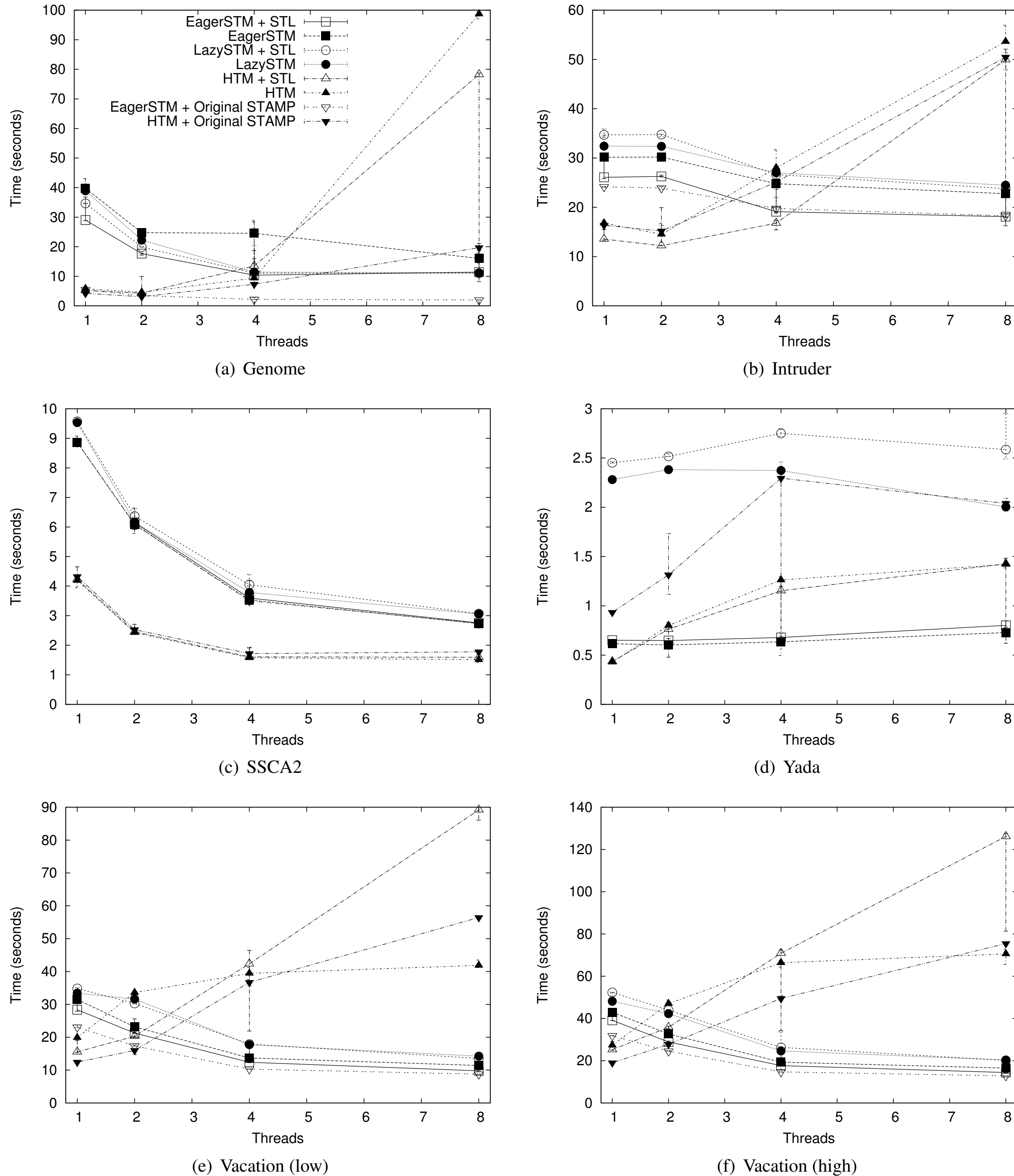
To gain further insight, we ran a red-black tree microbenchmark. STAMP's custom red-black tree is optimized for scalability. In particular, it does not support a `size` function. In contrast, the STL `std::set` object, which is also a red-black tree, requires every insertion and removal to modify a per-instance counter, in order to allow $O(1)$ determination of the size of the data structure.

We were surprised that this bottleneck did not affect Vacation's "+ STL" performance. Subsequently, we ran a red-black tree microbenchmark, in which threads performed lookups, insertions, and removals with equal probability, in a tree with 8-bit keys. Figure 5 compares a scalable red-black tree (RBTree) against `std::set` for this workload.

While the small size of the tree ensures many conflicts between operations, we still see good scaling for the custom tree on Westmere. In contrast, `std::set` hardly scales at all. Considering that the shared counter bounces between cores on every modification to the tree, this is not a surprise. In additional tests, we found that higher lookup ratios and larger key ranges led to some scaling for the STL tree, though still not close to the performance of the bottleneck-free tree. On Haswell, the story is even more dire: neither tree scales at all for the reported configuration. Again, for larger trees and higher lookup ratios, the custom tree is able to achieve some scaling. However, in this case there are two culprits: shared counters, and an apparent pathological interaction between HTM and the system allocator. Further exploration of this problem is left as future work.

6. Other Tools for TM Researchers

We briefly summarize two additional components of our new tool suite. First, we include a transactional version of memcached [22], and a version of PARSEC [2] that is fully compatible with the Draft C++ TM Specification [1]. This includes a version of transaction-



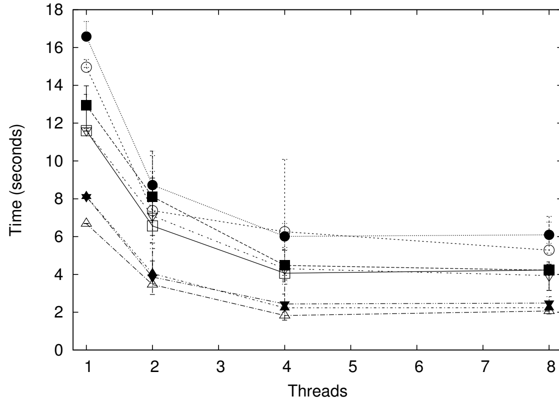
(a) Genome (b) Intruder (c) SSCA2 (d) Yada (e) Vacation (low) (f) Vacation (high)

Figure 3: STAMP results on Haswell system (1/2)

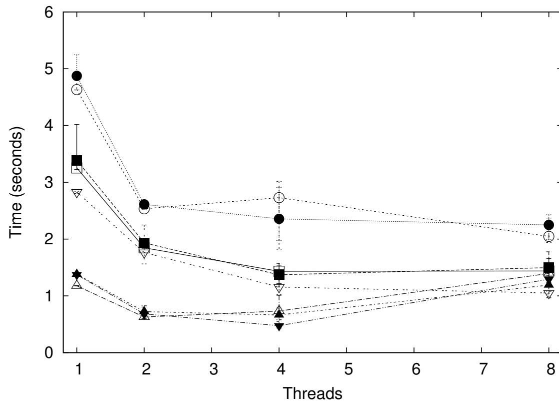
friendly condition variables. We believe that providing all of these tools in one place will make TM researchers more productive.

Secondly, we provide an extension to Intel's Threaded Building Blocks [12] that provides two new templates: an unordered speculative loop, and an ordered speculative loop. The former executes loop iterations in any order, running each as a transaction. This is well-suited for programs where nondeterminism is acceptable, but there may be conflicting accesses between iterations. This mechanism can use any TM implementation, without modification.

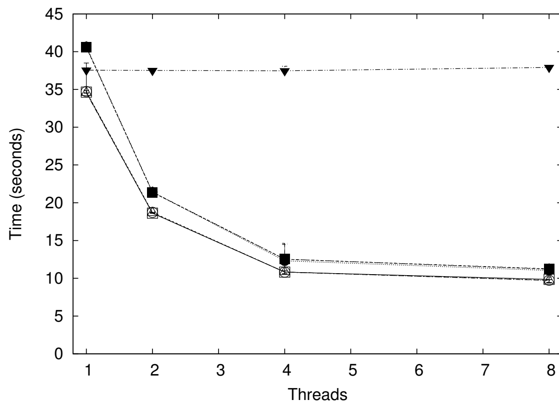
Our ordered speculative loop extends GCC-TM with a new function, `set_range`, which tells a TM implementation both (a) where a per-thread TBB range object can be found, and (b) the union of all ranges that will run for a given loop nest. We provide a set of custom TM implementations (based on eager software TM, lazy software TM, hardware TM, and the SpLIP algorithm [20]), which allow ordered transactional speculation [24] over arbitrary sequential loops. While this work is preliminary, it highlights the benefit of our simplified GCC environment: we can create a TBB



(a) KMeans (low)

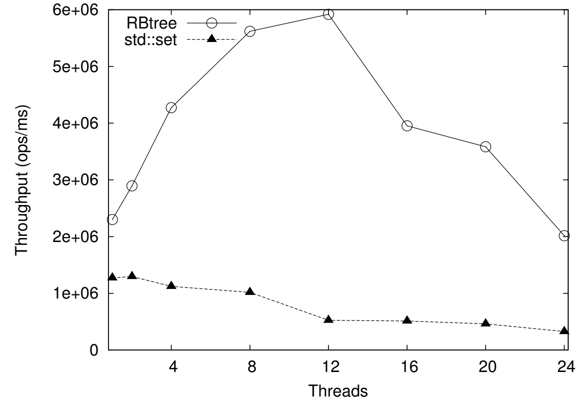


(b) KMeans (high)

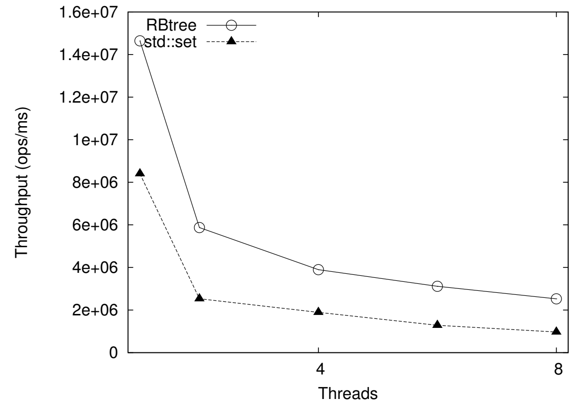


(c) Labyrinth

Figure 4: STAMP results on Haswell system (2/2)



(a) Westmere



(b) Haswell

Figure 5: Red-black tree micro-benchmark. Westmere used the EagerSTM runtime, Haswell used HTM.

template and connect it to a TM implementation in under 100 lines of code.

We are hopeful that, by providing our toolchain as an open-source repository, other researchers will be encouraged to contribute their programs and tools. To achieve consistency among tools and programs, some degree of porting will be required. However, we hope that it will ultimately be possible to include other benchmarks (e.g., AtomicQuake [26], SynQuake [17], and RMS-TM [15]) and tools (e.g., Lev's debugger [16], Gottschlich's visualizer [10], and Zyulkyarov's profiler [27]).

7. Conclusions

The third decade of TM research is upon us. This decade holds great promise, in that we have, at long last, real and production-quality tools at our disposal. With these tools comes a responsibility, as a community, to hold our research to a higher standard, especially when making claims about how results will apply to the real world. We believe that benchmarks should use the STL (and fix it, if scalability issues persist); new algorithms should be compatible with how real TM compilers work; and all researchers should have access to a broad array of benchmarks.

To support our position, we presented a TM-compatible STL (with discussion of its limitations), better implementations of lazy software TM for GCC, and additional tools to both aid in the evaluation of current systems, and encourage transactional researchers to branch out to new areas. Recognizing that our efforts are hardly the only in this area, we encourage other developers of core TM libraries and language support to share their work, so that we may create a clearing house of tools and benchmarks to encourage transactional memory's best decade yet.

Acknowledgments

We thank Jonathan Wakely for many discussions about GCC's implementation of `std::string`. This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. Any opinions, findings, and

conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, ON, Canada, Oct. 2008.

[3] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *Proceedings of the 27th ACM Conference on Programming Language Design and Implementation*, June 2006.

[4] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, Mar. 2011.

[5] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[7] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[8] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[9] Free Software Foundation. Transactional Memory in GCC, 2012. http://gcc.gnu.org/wiki/TransactionalMemory.

[10] J. Gottschlich, M. Herlihy, G. Pokam, and J. Siek. Visualizing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, Sept. 2012.

[11] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[12] Intel Corporation. Threaded Building Blocks. Available as www.threadingbuildingblocks.org.

[13] Intel Corporation. Intel Architecture Instruction Set Extensions Programming (Chapter 8: Transactional Synchronization Extensions). Feb. 2012.

[14] G. Kestor, L. Dalessandro, A. Cristal, M. Scott, and O. Unsal. Interchangable Back Ends for STM Compilers. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[15] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[16] Y. Lev. *Debugging and Profiling of Transactional Programs*. PhD thesis, Brown University, 2010.

[17] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[19] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[20] C. E. Oancea, A. Mycroft, and T. Harris. A Lightweight In-Place Implementation for Software Thread-Level Speculation. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[21] W. Ruan, Y. Liu, and M. Spear. STAMP Need Not Be Considered Harmful. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Mar. 2014.

[22] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, Mar. 2014.

[23] M. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[24] C. von Praun, L. Ceze, and C. Cascaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[25] R. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. Lee. 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*, Munich, Germany, June 2008.

[26] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

[27] F. Zyulkyarov, S. Stipic, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*, Vienna, Austria, Sept. 2010.