STAMP Need Not Be Considered Harmful [*]

Wenjia Ruan, Yujie Liu, and Michael Spear
Lehigh University
{wer210, yul510, spear}@cse.lehigh.edu

Abstract

The STAMP benchmark suite has not seen any updates in 5 years. During that time, language-level support for Transactional Memory (TM) has arrived, in the form of a draft specification for C++ and compiler support. In addition, there is now commodity hardware support for TM. The properties of STAMP, however, do not always match with the emerging consensus on how hardware and software transactions should be programmed, leading to questions of relevance.

We take the position that STAMP should not be considered harmful, and that the TM research community should embark upon a disciplined effort to produce up-to-date benchmarks, using STAMP as a starting point. To this end, we repair several flaws in STAMP and discuss their performance impact.

1. Introduction

The STAMP benchmark suite [6] is the most popular vehicle for evaluating hypothesis related to transactional memory (TM) [5]. Uses include the evaluation of hardware implementations, software implementations, contention managers and schedulers, compilers, and tools.

While STAMP filled a void, enabling apples-to-apples comparison between TM systems, there have always been some deficiencies. For example, the "Bayes" benchmark exhibits nondeterministic behavior; the "Yada" benchmark crashes for lazy software TM implementations; and the "Labyrinth" benchmark behaves differently with hardware TM than with software TM.

Before undertaking to fix these benchmarks, one must consider the context in which STAMP was created: its benchmarks were developed in 2006-2008, a time when no major vendor had yet committed to shipping a TM product, and during which many researchers outside the community wanted evidence that TM could be used for more than simple data structures. Consequently, STAMP benchmarks represent ideal workloads, intended to showcase TM, rather than representations of distinct and relevant classes of real-world workloads. Coupled with the fact that STAMP has seen no major revision in half a decade, one must ask if its continued use is beneficial to the community, or if STAMP should be considered harmful.

In this paper, we argue that STAMP need not be considered harmful. To justify our position, we present a set of characteristics that we believe a TM benchmark suite must provide, and we introduce several modifications to STAMP which remedy its most glaring problems. In the process, STAMP becomes a workload that (a) highlights successful approaches to transactional programming, (b) morphs into a set of programs more representative of what real-world programmers might write, and (c) can serve as a portion of a broader TM benchmark suite.

[*] This work was supported in part by the National Science Foundation under grants CNS-1016828, CCF-1218530, and CAREER-1253362.

While our work is in a preliminary state, we believe it crucial to share it with the TM research community as a first step toward forging a consortium to standardize a TM benchmark suite. As a result, this manuscript is partly a technical document, partly a position paper, and partly a call for participation. We begin by describing the current state of STAMP benchmarks (Section 2). In Section 3, we highlight the requirements which we believe ought to be satisfied by a TM benchmark suite. Section 4 then discusses the steps we took to move STAMP toward this goal. Section 5 presents performance results, which aid in determining whether our modifications help STAMP to conform to our requirements. Section 6 discusses next steps that we recommend in order to build momentum toward the creation of a suitable benchmark suite for TM. Section 7 concludes.

2. STAMP Benchmarks

STAMP consists of 8 applications, which differ greatly in how they use transactions. In this Section, we briefly describe each benchmark. These descriptions are largely based on those in [6].

Bayes The Bayes benchmark is fundamentally a graph traversal. In Bayes, an iterative hill-climbing strategy is used to learn the structure of a network, by computing the most likely structure given a set of observations. Transactions spend most of their time operating on a directed acyclic graph and a balanced tree. In general, transactions have large memory footprints (both for reading and writing). This benchmark exhibits nondeterminism: the ordering of commits among threads at the beginning of an execution can dramatically affect the execution time of the benchmark.

Genome This benchmark employs string matching to reconstruct a genome sequence from a set of DNA segments. Execution is in phases, with a large period of time during which most transactions are read-only. Transactions' access patterns are not particularly large, though non-trivial. There are two explicit phases in Genome, with the latter phase having a large implicit read-only phase for some inputs.

Intruder Intruder uses transactions to replace coarse-grained synchronization in a simulated network packet analyzer. The analyzer scans each packet to detect suspicious signatures. The primary shared data structures are a queue, for storing incoming packets, and a dictionary (a balanced tree) storing sets of packets that ought to be delivered together. The benchmark uses a pipeline, where the third phase does not use transactions.

KMeans This is an iterative clustering algorithm, in which transactions are used to protect the update of cluster centers during each iteration. The data points being clustered are partitioned among threads, resulting in relatively infrequent transactions, which are typically small. Transactions serve two roles: first, to avoid the use of locks when updating cluster centers, and second, to atomically apply a thread's updates to the set of cluster centers.

Labyrinth Labyrinth is an implementation of the Lee-TM circuit-routing benchmark [2]. The main data structure is a three-dimensional array, and threads must find non-overlapping paths to connect as many start and end points as possible. Threads do this by first copying the entire array to private memory, then computing a path, then using a transaction to update the grid, if and only if the path remains unoccupied since the time of the copy.

SSCA2 This benchmark contains four kernels, of which only the first is commonly run in TM experiments. The kernel involves performing small read-modify-write operations (with fewer than 5 reads and fewer than 4 writes per transaction) in order to build a directed, weighted multi-graph. The purpose of transactions is to avoid deadlocks that could occur when using fine-grained locking: since the set of locations to access is dynamically determined, deadlock is possible even though threads rarely access the same locations at the same time.

Vacation Vacation simulates online transaction processing. The underlying database is simulated via a set of trees, and transactions must access multiple trees to perform an operation. There are three classes of operations, and contention can be tuned by varying the ratio of transactions that perform large updates.

Yada In Yada, threads cooperate to perform a Delaunay refinement, using Ruppert's algorithm. An initial mesh is provided as input, and threads iterate through the triangles of the mesh to identify those whose minimum angle is below some threshold (i.e., they find triangles that are "too skinny"). When such a triangle is found, the thread adds a new point to the mesh, and then re-triangulates a region around that point, to produce a more visually appealing mesh. As in SSCA2, the set of locations accessed by a transaction are determined dynamically, which makes fine-grained locking difficult.

These benchmarks represent a diverse set of potential applications. The underlying parallelism of the workloads includes pipeline, worklist, and map patterns. Transaction durations vary considerably, as do the amounts of data read and written by transactions. In addition, benchmarks differ in the amount of contention and conflict they experience.

3. Requirements

A good benchmark must satisfy three high-level goals: it must test an important characteristic of the system, it must be *realistic*, and it must be *regular*. As discussed in Section 2, the first of these goals is handled well by STAMP. We expand upon the other two terms, placing them in the context of modern TM systems, and then evaluate STAMP's compatibility with them.

3.1 Realism

For a benchmark to be realistic, it must reflect some real-world problem, such that practitioners can trust that when a modification to a system (such as a CPU, compiler, or run-time library) improves the performance of the benchmark, it will improve the performance of many similarly-designed real-world programs. Additionally, the benchmark must be written in the same manner as a real-world program, and it should lack extraneous material that is irrelevant to the evaluation being performed.

Clearly the STAMP benchmarks satisfy the first requirement. Indeed, real-world problems were the primary motivation for developing the specific applications that comprise STAMP. However, STAMP does not resemble real-world code, and could benefit from some housekeeping. The main problem is that STAMP uses a library interface to instrument code, and the library interface provides functionality that is not supported by hardware TM, and that is also not incorporated into the proposed C++ TM specification.

Uninstrumented Accesses: When STAMP was created, there was no agreed-upon interface with which compiler writers could provide language-level TM support. Consequently, STAMP uses a library interface, with four core functions for beginning and ending transactions, and for reading and writing data. These functions are hidden behind macros, which are implemented differently depending on the execution context (hardware TM, compiler supported TM, library TM). At the current time, however, commercial compilers from Intel and IBM, as well as open source compilers from the GNU and LLVM projects, all provide some manner of language-level TM support. In all cases, this core interface is supported. In the case of GCC, it is straightforward to implement new TM algorithms within the GCC code base (each individual TM algorithm is contained in a single source file in the `libitm` source folder). Consequently, STAMP no longer needs to use manual instrumentation to ensure broad accessibility.

This is not merely a cosmetic change. First, we discovered that in the Yada benchmark, there is a write-before-read to shared memory in which the write is instrumented, but the read is not. When using TM with write-back, the uninstrumented read fails to see the preceding write, resulting in a crash. Second, the macros dictate sizes for data types, and these sizes are not always correct (e.g., when switching between 32-bit and 64-bit compilation). Third, a conservative compiler might instrument accesses at different granularities. In our experience, the compiler may generate calls to 128-bit SSE instructions, or may read individual bytes. Neither of these access sizes occur otherwise in STAMP. We are also concerned that STAMP's continued use of a library API encourages researchers to cut corners and implement a minimal TM when performing research, rather than meeting all the requirements of a production system. This makes comparison to a production-quality TM invalid.

Proposal: STAMP must transition to C++.

Impact: The conversion from C to C++ will have little to no impact on the behavior of STAMP applications. The only noteworthy consequence is increased compiler warnings, due to the more stringent requirements of the C++ language. For example, instances in which code assumes a 32-bit or 64-bit environment produce warnings with C++, but not with C. When compiled as a C program, these incorrect assumptions can produce incorrect results in 64-bit environments.

Proposal: STAMP must abandon its library interface in favor of a compiler-based interface. We recommend using the Draft C++ TM Specification [1].

Impact: The consequences of this change are discussed in the remainder of this section.

Pure Functions: The Draft C++ TM Specification does not allow for "transaction_pure" functions. Such functions, though called from a transactional context, are not instrumented. Unfortunately, in STAMP there is widespread use of tm_pure functions. Uses include calls to library code where the parameters are all thread-local, and comparators used by data structures. Again, we found that for lazy TM, these functions could cause errors. Furthermore, by hand-encoding the most optimized instrumentation, STAMP currently affords limited opportunity to assess the value of certain compiler optimizations [3, 8].

Note that we are not arguing that support for pure functions should be removed from compilers, but rather that they should not be used in STAMP unless it is absolutely unavoidable. Optimizing performance via pure functions, in a manner that breaks write-back STM, is clearly not acceptable. The community should decide whether using pure functions to avoid limitations in existing compilation systems (e.g., the lack of transaction-safe math libraries) ought to be allowed.

Proposal: STAMP should not use tm_pure functions.

Impact: Removing tm_pure functions will ensure that STM will not risk introducing races, that lazy STM will work correctly, and that HTM and STM will not exhibit different conflict patterns. However, STM performance will be penalized in some circumstances.

Self-Abort: Since TM implementations typically allow for transactions to roll back, it is appealing to allow the programmer to invoke the roll-back machinery manually, via a linguistic construct for self-abort with immediate restart. Such a mechanism is difficult to use correctly [7], and the most recent version of the Draft C++ TM Specification argues for "cancellation" rather than self-abort with automatic restart. STAMP, however, uses the older "restart" mechanism.

Upon further analysis, STAMP's use of "restart" is extremely unorthodox. In Vacation, "restart" appears on code paths that cannot be reached if the underlying TM system is opaque [4]. Since opacity is an assumed property of compiler-based TM and hardware TM, these restarts are unnecessary. In Yada and Labyrinth, "restart" is used for explicit speculation. In Labyrinth, the explicit speculation includes intentional uninstrumented reads. As we will show, these reads, and indeed all explicit speculation, can be encoded via control flow outside of the transaction, making self-abort unnecessary.

Proposal: STAMP should not use self-abort, and should minimize its use of explicit transaction cancellation.

Impact: Yada and Labyrinth will require redesign, since they use self-abort to manipulate control flow in unconventional ways.

Manual Cloning: If a function is called both from transactional and nontransactional contexts, then two versions of the function must appear in the program binary: one optimized for the nontransactional context, and the other carrying suitable instrumentation so that it can be rolled back when called from a software transaction. Without the second version of the code, a system will be unable to use most STM algorithms.

In the C++ specification, both versions of the executable code are generated from the same source code implementation of the function. This ensures that programmers need not learn multiple names for the same function, and that they do not accidentally fail to modify one source file when maintaining code. With explicit cloning, the programmer can perform I/O or call an unsafe function from one implementation, but not from the other. This, in turn, means that TM implementations that can use the uninstrumented function (e.g., hardware TM) may not behave the same as those that cannot. Note, too, that manual cloning nearly doubles the size of several source files.

Proposal: STAMP should rely on the C++ compiler for cloning.

Impact: STM performance may be penalized, but the overall code size will be reduced by over 20%.

Reimplementation of Standard Libraries: Lastly, to ensure instrumentation of standard libraries, STAMP re-implements any standard library calls made from within transactions. As C++ TM support matures, this should become unnecessary. Unfortunately, at this time reimplementation appears necessary [10]. However, STAMP lacks the mechanism to selectively re-implement only those libraries that do not have transaction-safe variants. Furthermore, the reimplementation should be reflective of real-world code, ideally by providing minimally-altered source code for the most recent implementations of the missing files. The most natural way to provide this support is at build time, via a configuration script that evaluates each library function individually, and links only the STAMP-supplied versions of those functions that are unavailable.

It is important to note that this change must include all data structures used by STAMP. Currently, STAMP implements its own map, list, and queue data structures. These should be replaced with transactional versions of the C++ standard template library.

Proposal: STAMP's reimplementation of standard libraries and data structures should be minimal, and controlled by the build system.

Impact: This change will require the re-implementation of functions that are currently called *in an unsafe manner* within STAMP. It will also require the transactionalization of portions of the Standard Template Library. Upon completion, these contributions are likely to be valuable to a broad community.

The above recommendations will help ensure that STAMP is both representative of real-world code, and implemented in a manner that is reflective of real-world programming practices. A final point is that STAMP benchmarks must be streamlined. Currently, only one benchmark is deficient in this area: SSCA2. In SSCA2, macros govern whether "scalable data generation" is available or not. There are also multiple kernels that can be tested, of which only Kernel #1 is typically reported in publications. We believe that SSCA2 should be changed to remove excess code, so that it only contains the most popular configuration.

3.2 Regularity

A benchmark must have regular and predictable behavior. In STAMP, the only benchmark affected by this requirement is Bayes. Its execution time is highly dependent on the interleaving of transactions from multiple threads. It may ultimately be best to remove Bayes from STAMP, since this interleaving cannot be made deterministic via replay of an input trace. Note that KMeans, Yada, and Labyrinth employ input traces.

Ideally, the benchmarks should produce a "correct answer" that can be verified; even if not, the behavior of the benchmark, when run multiple times under the same configuration, should not vary. To justify this point, we recount a situation in which GCC version 4.3.2 on the Solaris OS, when compiling Vacation, elided a function call, resulting in more than 20% of transactions executing in read-only mode. While this bug was not due to a property of STAMP, we only discovered the bug because of different performance results when running STAMP+RSTM on Solaris and x86 platforms! We believe that adding a verification step to STAMP is a necessity.

4. Fixing STAMP

We now outline the modifications we made to STAMP. Our goal was to adhere to the positions described in Section 3 as much as possible.

4.1 Basic Modifications

Our first round of modifications entailed the removal of `TM_PURE` annotations from functions. We also annotated the function pointers for the comparators given to the list and map collections, so that they were transaction-safe. We then eliminated any manually cloned functions, so that all instrumentation was generated by the compiler, from uninstrumented source code. This removed a STAMP optimization in which transaction descriptors were manually passed on the stack.

This step identified the following collections as being manually cloned: list, map(rbtree), vector, queue, and bitmap. These data structures are implemented in an ad-hoc fashion, which may be dangerous or misleading: The rbtree is based on an implementation from researchers at Oracle, which is fine-tuned for transactions; the queue has a fixed size, yet has no condition synchronization to handle enqueuing into a full queue or dequeueing from an empty queue. All of these collections should ultimately be replaced with standard C++ library code.

In this effort, we also identified a number of math functions that were not transaction-safe. These functions are truly *pure*: they are side effect-free, and their return value is determined only by their (private, scalar) inputs. We exposed the implementation of these functions to the compiler so that it could produce transaction-safe clones. This effort identified that the Draft C++ TM Specification should require these math functions to be transaction-safe.

In all, these basic modifications affected all benchmarks except SSCA2.

4.2 Removing Restart

As discussed earlier, removing calls to `TM_RESTART` from Vacation was trivial: the calls were on code paths that cannot be live for an opaque TM. However, Yada and Labyrinth required additional attention.

In Labyrinth, a transaction would (a) perform an uninstrumented `memcpy` of the main data structure (a matrix), (b) perform an uninstrumented traversal of the copy to compute a path, and then (c) update the main data structure via a loop of the following form:

```
for (p : points)
    if (TM_READ(p.available))
        TM_WRITE(p.available = false);
    else TM_RESTART();
```

This code conflates manual speculation and transactional rollback. A simple transformation could eliminate both. The above loop simply became:

```
flag = true;
for (p : points)
    if (!TM_READ(p.available))
        flag = false;
        break;
if (flag)
    for (p : points)
        TM_WRITE(p.available = false);
```

This modification performs all reads before their dependent writes. Consequently, restart is not necessary: we can simply commit a read-only transaction, instead of aborting and rolling back a writing transaction. By coupling this change with some additional control flow outside of the transaction, we eliminated the explicit racy read of the matrix (replacing it with a read-only transaction), and then performed the data structure updates in a small transaction.

The explicit speculation in Yada is inherent to the algorithm's design. While we could not remove speculation, we were able to replace `TM_RESTART` with `transaction_cancel`. As with Labyrinth, coupling this change with some simple control flow outside of the transaction was sufficient to update Yada to compatibility with the current Draft C++ TM Specification. However, if C++ does not include some form of cancellation or self-abort, then Yada will require re-implementation.

4.3 Summary

Though the sizes of the source code diffs are quite large, our modifications to STAMP were relatively minor. We removed hundreds of lines of duplicate code, instead allowing the compiler to generate clones. We refactored self-abort in two benchmarks, annotated function pointers as transaction-safe in 7 benchmarks, and added safe versions of a handful of math functions. We only required atomic transactions, since no transaction in STAMP performs I/O.

In the longer term, we expect the compiler and run-time system to provide safe math functions, which will further reduce the size of STAMP. Similarly, the collections and associated functions (to include qsort, which is called by transactions in Bayes) which STAMP currently provides should ultimately not be required, since the run-time system should provide transaction-safe collections.

5. Performance Evaluation

We now discuss the performance of our modified STAMP benchmarks. The primary purpose of this evaluation is to assess regularity and predictiveness of the benchmarks. To that end, we added a new TM implementation to GCC, which uses commit-time locking and write back (essentially a privatization safe version of PatientTM [9]).

We compare the default (eager/undo) software TM provided by GCC (ml_wt), our lazy software TM (lazy_wb), and the hardware TM available on recent Intel CPUs, as accessed through the GCC interface. We performed experiments on two machines. "Westmere" experiments were performed on a 6-core/12-thread Intel Xeon X5650 CPU running at 2.67GHz; "Haswell" experiments used a 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz. Both machines were running Ubuntu Linux 13.04 with kernel version 3.8.0. All benchmarks were compiled using GCC, version 4.8.0. The compiler was configured to use its ml_wt software TM algorithm on Westmere, and to use its HTM algorithm on Haswell. All code was compiled at –O2 optimization level, as –O3 led to the use of 128-bit SSE instructions that were incompatible with our lazy TM algorithm. All experiments reported in this paper are the average of five trials.

5.1 Software TM Results

To assess the effectiveness of our modified STAMP as a benchmark for evaluating software TM implementations, we compare the original STAMP code, running with GCC's eager TM, our revised STAMP with GCC's eager TM, and our revised STAMP using a lazy TM algorithm. Figure 1 presents performance on each of the 8 benchmarks, and includes two charts for Vacation and KMeans, to reflect the two different recommended contention levels.

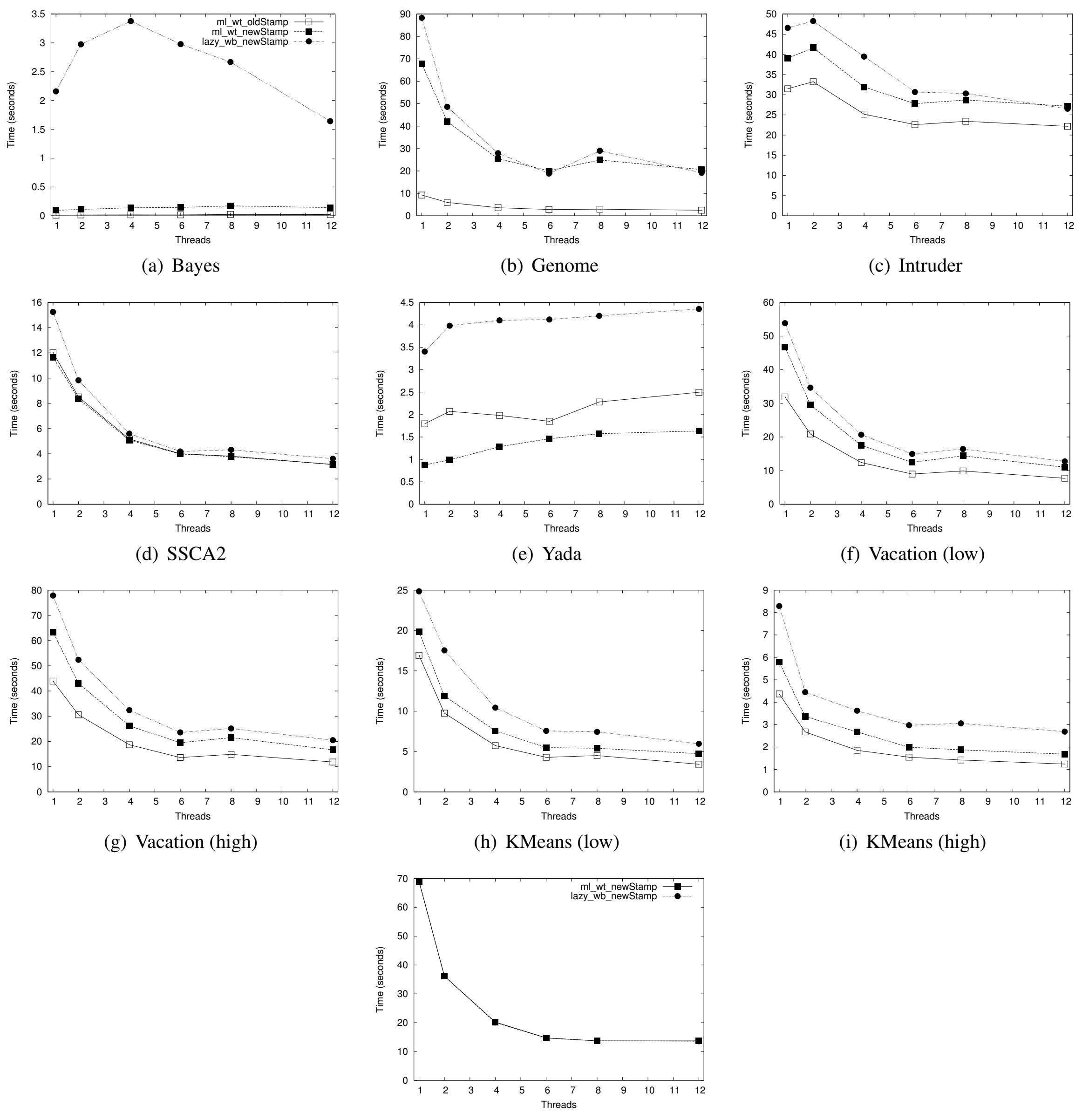
For the most part, the results are uninteresting. Our modifications make it possible to use lazy TM algorithms, but otherwise Intruder, SSCA2, Vacation (both levels), and KMeans (both levels) are without surprises: our changes add overhead to STAMP, which is reflected by the eager TM experiencing a slowdown at all thread counts. Our lazy TM, which uses an unbalanced binary search tree for its write set, instead of a more efficient hash table, incurs overhead proportional to the size of transactions.

Genome's execution time increases significantly, due to the compiler instrumenting more memory accesses (particularly on account of calls to strcmp). Otherwise, Genome remains a predictable benchmark. Labyrinth performs as expected, with run times and behavior comparable to previous published results. Note, however, that Labyrinth was unable to execute with the baseline GCC algorithm: calls to transaction restart (not cancel) led to run-time errors.

Yada and Bayes exhibit unexpected behavior. Both show high overheads for lazy TM, due to write set lookup and write-back overheads for large transactions. In Yada, the new version, with more instrumentation, outperforms the older version when run with an eager software TM. The savings appears to come from better inlining of math functions in our revised code. We intend to remove this artifact in the final version of this document, as part of redesigning the build system.

In Bayes, expensive calls to `qsort` from within a transaction, which become instrumented in our GCC, substantially perturb the benchmark. Other sources of overhead include recursive computation of log likelihood, which became instrumented, and calls to math libraries. Overheads for lazy TM were unacceptably high, leading us to decrease the input problem size in an attempt to achieve execution times that were not embarrassingly slow compared to locks. On the other hand, the execution times showed little variance among trials.

If we only cared about software TM, it would appear that with our modifications, STAMP, with Bayes removed, remains an inter-



(a) Bayes (b) Genome (c) Intruder

(d) SSCA2 (e) Yada (f) Vacation (low)

(g) Vacation (high) (h) KMeans (low) (i) KMeans (high)

(j) Labyrinth

Figure 1: STAMP results on Westmere system

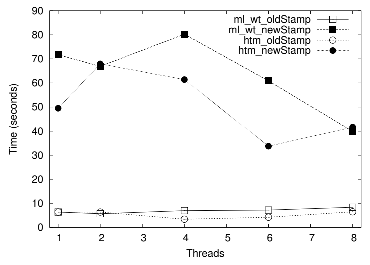
esting set of workloads for a benchmark: all are now compatible with the C++ specification, and most exhibit reasonable scalability with general-purpose TM libraries.

5.2 Hardware TM Results

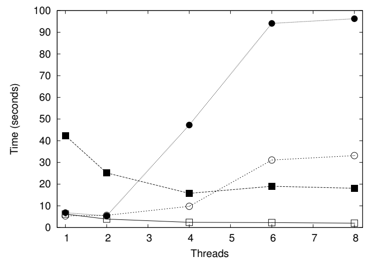
The situation is much murkier with hardware TM (Figure 2). Our expectation was that hardware TM would reduce latency, but otherwise exhibit scalability similar to the baseline software TM. We anticipated that this trend could taper after 4 threads, due to symmetric multithreading (SMT): when the L1 cache is shared among threads, hardware TM capacity drops, increasing fallback to serial mode.

SSCA2 met our expectations. The other benchmarks did not, for a variety of reasons. In Labyrinth, transactions are simply too rare to observe a speedup for small hardware transactions relative to software transactions. In KMeans, hardware TM suffered at 8 threads: presumably this is due to preemption and SMT-based cache sharing causing hardware transactions to fall back to serial mode.

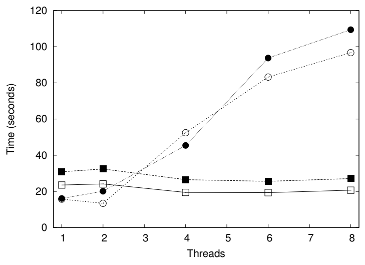
On Vacation, hardware TM performance degraded significantly for multithreaded runs. As far as we can tell, this degradation was due to contention among transactions that caused livelock. To overcome the livelock, the TM fell back to serial mode, but not before wasting significant work. To confirm this hypothesis, we ran the original STAMP, with relaxed transactions and no annotations, using the hardware TM algorithm. We observed exactly the same trend, though in the low contention case the livelock did not manifest until 3 threads. The same phenomenon was observed on Intruder and Genome.



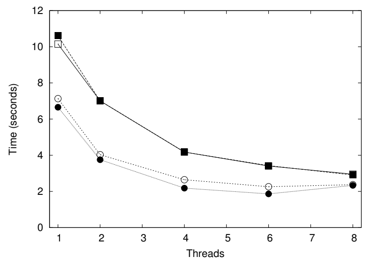
(a) Bayes



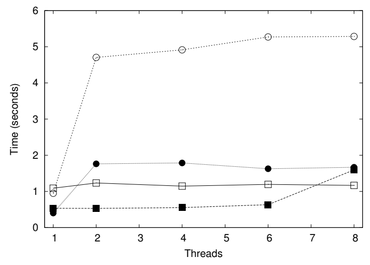
(b) Genome



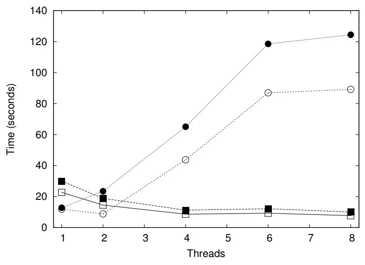
(c) Intruder



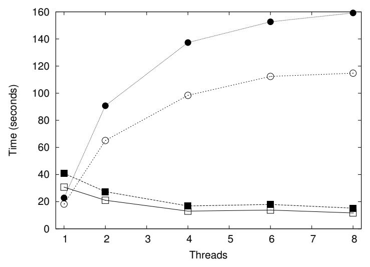
(d) SSCA2



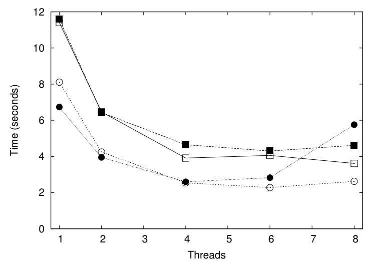
(e) Yada



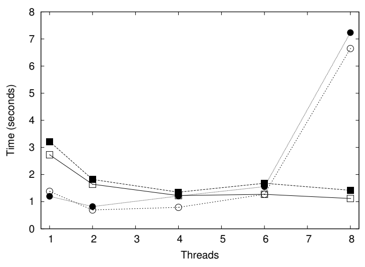
(f) Vacation (low)



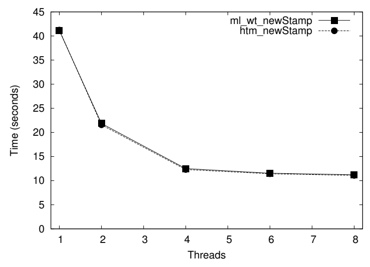
(g) Vacation (high)



(h) KMeans (low)



(i) KMeans (high)



(j) Labyrinth

Figure 2: STAMP results on Haswell system

As with software TM, Yada shows improvement after our updates. We expect that this artifact is due to inlining, and can be removed, leaving Bayes as the sole benchmark for which performance remains unpredictable.

Given that the performance anomalies we observe can all be explained by properties of the hardware TM, we conclude that most of STAMP remains an interesting vehicle for testing TM implementations.

6. Discussion

It is perhaps still too early to craft a standard benchmark suite for TM. In particular, the Draft C++ TM Specification has seen major revision over the last year, and continues to evolve. Without a clear picture of what the language exposes to the programmer, decisions about what functionality must be evaluated by a benchmark may be overcome by events. Nowhere is this more significant than with self-abort or cancellation. If some form of self-abort is part of the specification, then a good benchmark suite must use it in a manner that allows for the evaluation of its performance. If self-abort is not part of the specification, then STAMP should not use it.

Another key question is whether peculiarities observed today can be remedied by changes to software systems. If the behavior of Vacation is fundamental to the sorts of conflict detection and capacity properties that are built into modern hardware TM, then Vacation becomes a less interesting benchmark: only architecture researchers have the ability to improve its performance under hardware TM. If, on the other hand, software can improve performance (perhaps via upcoming support in GCC 4.9 for a hardware TM fast-path), then these benchmarks will be valuable to researchers in hardware and software.

It is important to note, too, that the specification does not simply influence the benchmarks; benchmarks may also influence the specification. The most pressing example here is contention management. More than a dozen novel approaches to transaction scheduling and contention management have been proposed over the last decade, many in the context of STAMP. It is appropriate for STAMP benchmarks to demonstrate some degree of sensitivity to contention management parameters, and how that sensitivity is shown may influence how a future C++ TM Specification exposes contention management to the program.

Additionally STAMP should emphasize relaxed transactions or atomic transactions. In its current form, STAMP can use either interchangeably, except for Yada, where the use of cancellation necessitates atomic transactions. As these distinct linguistic constructs become more precisely defined, it will become important for TM benchmarks to expose differences in how different systems implement each type of transaction, so that those implementations can be evaluated.

Lastly, the relationship between STAMP and HTM must be given further consideration. Questions include whether STAMP should provide facilities or behaviors that stress Hybrid TM behaviors, how to address fallback to a single global lock for HTM transactions that cannot fit in the cache, and what importance, if any, should be given to progress guarantees. It is likely that new benchmarks will need to be created to capture the difference between the research HTM proposals that were popular at the time of STAMP's publishing, and the "best-effort" HTM systems available today.

We believe that the right "first step" is embodied by this paper: updating the most popular TM benchmarks, so that they adhere to the latest standards for how transactional code ought to be written. By making our work open-source, we hope to encourage contribution of patches and additional benchmarks, and to create an opt-in opportunity for others to join our effort to improve upon the state-of-the-art in benchmarking for transactional memory.

7. Conclusions and Future Work

In this paper, we outlined a set of properties that a good benchmark should have, and discussed the effort required to achieve those properties in the STAMP benchmark suite. Our changes had a significant impact on performance, regularity, and variance for the STAMP benchmarks.

We are hopeful that this work spurs the formation of a working group to begin the process of transitioning from STAMP to a more broadly agreed-upon standard benchmark suite for transactional memory. Such a suite will need to adapt quickly to changes in the TM community, but we believe it will ultimately enable the community to perform better evaluation, and to ensure that research results can benefit practical and real-world code.

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] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, June 2008.

[3] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[4] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[5] 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.

[6] 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.

[7] V. Pankratius and A.-R. Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[8] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[9] 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.

[10] T. Vyas, Y. Liu, and M. Spear. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, Houston, TX, Mar. 2013.