STAMP: Stanford Transactional Applications for Multi-Processing

Chí Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun
Computer Systems Laboratory
Stanford University
{caominh, jwchung, kozyraki, kunle}@stanford.edu

***Abstract*—Transactional Memory (TM) is emerging as a promising technology to simplify parallel programming. While several TM systems have been proposed in the research literature, we are still missing the tools and workloads necessary to analyze and compare the proposals. Most TM systems have been evaluated using microbenchmarks, which may not be representative of any real-world behavior, or individual applications, which do not stress a wide range of execution scenarios.**

We introduce the Stanford Transactional Application for Multi-Processing (STAMP), a comprehensive benchmark suite for evaluating TM systems. STAMP includes eight applications and thirty variants of input parameters and data sets in order to represent several application domains and cover a wide range of transactional execution cases (frequent or rare use of transactions, large or small transactions, high or low contention, etc.). Moreover, STAMP is portable across many types of TM systems, including hardware, software, and hybrid systems. In this paper, we provide descriptions and a detailed characterization of the applications in STAMP. We also use the suite to evaluate six different TM systems, identify their shortcomings, and motivate further research on their performance characteristics.

I. Introduction

Multi-core chips are now commonplace in server, desktop, and even embedded systems. However, multi-core chips create an inflection point for mainstream software development. To benefit from the increasing number of cores per chip, application developers have to develop parallel programs and deal with cumbersome issues such as synchronization tradeoffs, deadlock avoidance, and races. In this setting, Transactional Memory (TM) [20] has surfaced as a promising technique to help with shared-memory parallel programs. With conventional lock-based multithreaded programming, programmers need to manually manage the concurrency among threads. With TM, programmers simply mark code segments as transactions that should execute *atomically* and *in isolation* with respect to other code. The TM system automatically manages synchronization issues without further burden on the application developer.

There have been many proposed designs for TM systems based on hardware (HTM) [6, 16, 29], software (STM) [13, 17, 19, 27, 34], and hybrid hardware/software techniques [8, 12, 25, 33, 36, 39]. Most evaluations of these systems have relied on microbenchmarks or parallel applications from benchmark suites like SPEComp [38] or SPLASH-2 [41]. While microbenchmarks are useful in targeting specific system features, they are not representative of how full applications will behave on TM systems. On the other hand, the benchmarks in SPEComp and SPLASH-2 are full applications, but they have been heavily optimized by an expert (typically as part of a computer science Ph.D. thesis) to minimize synchronization and communication across threads. Thus, converting their fine-grain lock-protected regions to transactions leads to programs that rarely use transactions, making it hard to evaluate the differences among TM systems. Furthermore, this behavior is not necessarily representative of how mainstream programmers will use transactions in their programs. The most appealing potential of TM for many programmers is the ability to write simple parallel code with frequent use of coarse-grain transactions that perform as well as code that has been carefully optimized to use fine-grain locks.

For a benchmark suite to enable a thorough analysis of a wide range of TM systems, it must have three key features. First, it must target a *variety of algorithms and application domains that can benefit from TM*. Second, it must cover a *wide range of transactional characteristics* such as transaction lengths and sizes of read and write sets. Moreover, the amount of time spent in transactions should be varied. Certain applications or data sets in the suite should generate cases where a significant portion of time is spent in transactions. This requirement allows the performance of TM systems to be evaluated with frequently-used, coarse-grain transactions. Finally, the benchmark suite must be *compatible with a large number of TM systems*, covering hardware, software, and hybrid designs.

To meet these three requirements, we have developed the *Stanford Transactional Applications for Multi-Processing (STAMP)*, a new benchmark suite designed specifically for evaluating TM systems. The STAMP suite consists of eight applications with 30 different sets of configurations and input data that exercise a wide range of transactional behaviors. Moreover, the STAMP code can run on a variety of hardware, software, and hybrid TM systems. The specific contributions of this work are as follows:

- We introduce STAMP, a new benchmark suite for transactional memory systems. We describe the algorithms and data structures, the parallelization strategy, and the use of transactions in each of the eight applications.
- We provide a transactional characterization of the STAMP applications. In particular, we measure the transaction length, the sizes of the read and write sets, the amount of time spent in transactions, and the average number of retries per transaction.

- To demonstrate the usefulness of STAMP for TM research, we use it to analyze the performance of six different TM systems: two variants each of hardware, software, and hybrid TM designs. The conventional wisdom in the TM community has been that eager versioning leads to better performance than lazy versioning and increasing amounts of hardware support for TM lead to significant performance improvements. Using STAMP, we show that these generalizations are invalid in certain cases and discuss the technical issues behind them.

The rest of this paper is organized as follows. Section II reviews related benchmark efforts, and Section III introduces our new TM benchmark suite. Sections IV and V present the experimental methodology and results, respectively. Finally, Section VI concludes the paper and comments on future work.

II. Related Work

Researchers and industry consortiums have created many benchmarks for evaluating parallel systems. Recently, some benchmarks for TM systems have emerged as well. This section reviews the scope, strengths, and shortcomings of these efforts with respect to evaluating TM systems.

A. Parallel Benchmarks

Three of the oldest and most widely used parallel benchmarks are SPLASH-2 [41], NPB OpenMP [22], and SPEComp [38]. More recently, domain-specific parallel benchmarks have also been created, such as BioParallel [21] for bioinformatics and MineBench [30] for data mining. These benchmark suites consist of several applications that span a variety of algorithms. Additionally, all of them utilize OpenMP as their programming model, except for SPLASH-2, which uses a Pthreads-like model via ANL macros.

PARSEC [4] is a very recent parallel benchmark that was created to address some of the shortcomings in the above-mentioned benchmarks. Through selection of the PARSEC applications, care was taken to use state-of-art techniques in a range of application domains not limited to just high-performance computing. The applications are written in either C or C++ and use either OpenMP or Pthreads to implement the parallelization.

Even though these benchmarks are useful in analyzing parallel systems in general, their applicability to TM systems is limited. For example, many of the applications in these benchmarks consist of regular algorithms that can be parallelized without stressing synchronization (i.e., just using barriers is sufficient). Furthermore, in order to achieve the highest possible performance, the irregular applications in these benchmarks have been carefully optimized by expert programmers to minimize the time spent in critical regions. Converting the critical regions in these applications to transactions, leads to programs with small transactions that are rarely used. Consequently, these benchmarks are unable to sufficiently stress the underlying TM system and do not generate most of the interesting cases of transactional behavior with respect to transaction length and frequency or sizes of read and write sets. Since manual tuning of the application code by an expert contradicts the primary goal of transactions memory (practical parallel programming for mainstream developers), usage of these benchmarks alone to evaluate TM systems may be misleading.

B. Transactional Memory Benchmarks

To better target TM systems, researchers have begun creating new workloads. Existing efforts can be grouped into two general categories: microbenchmarks and individual applications. TM microbenchmarks are typically composed of transactions that execute a few operations on a data structure like a hash table or red-black tree [12, 13, 25, 27, 34]. These microbenchmarks are easy to develop, parameterize, and port across systems. Furthermore, they are very useful for isolating particular cases of interest in the TM system. However, they are not representative of full applications. Full applications are likely to have transactions that consist of many operations across several data structures and may also include significant amounts of parallel or sequential work between transactions. Thus, the transactional characteristics of microbenchmarks and full applications may be significantly different.

A few larger workloads that target TM systems have been designed to address the disparity between microbenchmarks and full applications: Delaunay mesh generation [35], database management [14], BerkeleyDB [12], maze routing [40], and Delaunay mesh refinement and agglomerative clustering [24]. These applications represent realistic workloads and avoid the pitfalls of microbenchmarks. However, as they are all standalone and not part of a larger suite, their coverage of algorithms and transactional behaviors is limited. Moreover, only the first two of these applications are publicly available.

More recent work has focused on the creation of TM benchmark suites. Perfumo et al. [31] have created a suite of nine applications targeted at evaluating STMs. However, the applications are implemented in Haskell and are thus not directly compatible with the majority of STM implementations or HTM and hybrid TM simulation environments. Almost all of these applications are also microbenchmarks.

III. Stanford Transactional Applications for Multi-Processing (STAMP)

At present, no benchmark suite for TM covers a wide enough range of algorithms and application domains, stresses most cases of transactional behavior, and runs easily on many classes of TM systems. The *Stanford Transactional Applications for Multi-Processing (STAMP)* is the first benchmark suite to satisfy all of these requirements. STAMP consists of eight applications with 30 different sets of configurations and input data that exercise a wide range of transactional behaviors. Moreover, STAMP can run on a variety of hardware, software, and hybrid TM systems. STAMP is publicly available at `http://stamp.stanford.edu`.

A. Design Philosophy

The design of the STAMP benchmark suite follows three guiding principles to make it an effective and comprehensive tool for evaluating TM systems:

TABLE I: Summary of publicly available benchmark suites used to evaluate TM systems. The bottom half of the table lists benchmarks created specifically for analyzing TM systems. The *Breadth* column also lists the number of applications or kernels included in each suite.

Benchmark	Breadth	Depth	Portability
SPLASH-2 [41]	yes (12)	no	partial
NPB OpenMP [22]	yes (7)	no	partial
SPEComp [38]	yes (11)	no	partial
BioParallel [21]	partial (5)	no	partial
MineBench [30]	partial (15)	no	partial
PARSEC [4]	yes (12)	no	partial
RSTMv3 [27, 35]	no (6)	yes	yes
STMbench7 [14]	no (1)	yes	yes
Perfumo et al. [31]	yes (9)	yes	no
STAMP	**yes** (8)	**yes**	**yes**

1) **Breadth:** STAMP consists of a variety of algorithms and application domains. In particular, we favor ones that are not trivially parallelizable without synchronization, as they can benefit significantly (in terms of ease of programming) from TM's optimistic concurrency.
2) **Depth:** STAMP covers a wide range of transactional behaviors such as varying degrees of contention, short and long transactions, and different sizes of read and write sets. We also include applications that make frequent use of coarse-grain transactions and spend a significant portion of their execution time within transactions. This results in a balanced set of workloads that adequately stress the underlying TM system.
3) **Portability:** STAMP can easily run on many classes of TM systems, including hardware-based (HTM), software-based (STM), and hybrid designs. The code for all benchmarks is written in C with annotations to indicate both transaction boundaries and memory accesses that require instrumentation for software and hybrid systems. We have successfully run the suite on six TM systems, including hardware, software, and hybrid designs.

Table I summarizes how various parallel and TM benchmarks meet the three requirements discussed above. To the best of our knowledge, STAMP is the only one that satisfies all three requirements at this point. All the parallel benchmarks (upper half of Table I) are only partially portable with regard to TM systems as they lack annotations for generating STM read and write barriers. As for the TM benchmarks (lower half of table), RSTMv3 [27, 35] consists of mostly microbenchmarks and thus does not satisfy the breadth requirement.

B. Applications

STAMP currently consists of eight applications: `bayes`, `genome`, `intruder`, `kmeans`, `labyrinth`, `ssca2`, `vacation`, and `yada`. These applications span a variety of computing domains as well as runtime transactional characteristics such as varying transaction lengths, read and write set sizes, and amounts of contention. Table II gives a brief description of each benchmark.

TABLE II: The eight applications in the STAMP suite.

Application	Domain	Description
bayes	machine learning	Learns structure of a Bayesian network
genome	bioinformatics	Performs gene sequencing
intruder	security	Detects network intrusions
kmeans	data mining	Implements K-means clustering
labyrinth	engineering	Routes paths in maze
ssca2	scientific	Creates efficient graph representation
vacation	online transaction processing	Emulates travel reservation system
yada	scientific	Refines a Delaunay mesh

1) `bayes`: This application implements an algorithm for learning the structure of Bayesian networks from observed data. The algorithm uses a hill-climbing strategy that combines local and global search, similar to [11]. An adtree data structure [28] is used to achieve efficient estimates of probability distributions. The Bayesian network itself is represented as a directed acyclic graph, with a node for each variable and an edge for each conditional dependence between variables. Initially, the network has no dependencies among variables, and the algorithm incrementally learns dependencies by analyzing the observed data. On each iteration, each thread is given a variable to analyze, and as more dependencies are added to the network, connected subgraphs of dependent variables are formed.

A transaction is used to protect the calculation and addition of a new dependency, as the result depends on the extent of the subgraph that contains the variable being analyzed. Using transactions is much simpler than a lock-based approach as using locks would require manually orchestrating a two-phase locking scheme with deadlock detection and recovery to allow concurrent modifications of the graph. Calculations of new dependencies take up most of the execution time, causing `bayes` to spend almost all its execution time in long transactions that have large read and write sets. Overall, this benchmark has a high amount of contention as the subgraphs change frequently.

2) `genome`: Genome assembly is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. This program has two phases to accomplish this task. Since there is a relatively large number of DNA segments, there is often many duplicates. The first phase of the algorithm utilizes a hash set to create a set of unique segments. In the second phase of the algorithm, each thread tries to remove a segment from a global pool of unmatched segments and add it to its partition of

currently matched segments. When matching segments, Rabin-Karp string matching [23] is used to speed up the comparison.

Transactions are used in each phase of the benchmark. Additions to the set of unique segments are enclosed by transactions to allow concurrent accesses, and accesses to the global pool of unmatched segments are also enclosed by transactions since threads may try to remove the same segment. By using transactions for the reconstruction, we did not have to implement a deadlock avoidance scheme. Overall, the transactions in `genome` are of moderate length and have moderate read and write set sizes. Additionally, almost all of the execution time is transactional, and there is little contention.

3) `intruder`: Signature-based network intrusion detection systems (NIDS) scan network packets for matches against a known set of intrusion signatures. This benchmark emulates Design 5 of the NIDS described by Haagdorens et al. in [15]. Network packets are processed in parallel and go through three phases: *capture*, *reassembly*, and *detection*. The main data structure in the *capture* phase is a simple FIFO queue, and the *reassembly* phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session. When evaluating their designs for a multithreaded NIDS, Haagdorens et al. state that the complexity of the *reassembly* phase caused them to use coarse-grain synchronization. Thus, even though they attempted to exploit higher levels of concurrency, their coarse-grain synchronization resulted in worse performance.

In the TM version included in STAMP, the *capture* and *reassembly* phases are each enclosed by transactions. Hence, the code for each phase is as simple as that with coarse-grain locks but hopefully achieves good performance through optimistic concurrency. When operating on these data structures, this benchmark has relatively short transactions. It also has moderate to high levels of contention depending on how often the *reassembly* phase rebalances its tree. Overall, since two of the three phases are spent in transactions, this benchmark has a moderate amount of total transactional execution time.

4) `kmeans`: The K-means algorithm groups objects in an N-dimensional space into K clusters. This algorithm is commonly used to partition data items into related subsets. The implementation used is taken from MineBench [30] and has each thread processing a partition of the objects iteratively. The TM version add a transaction to protect the update of the cluster center that occurs during each iteration. The amount of contention among threads depends on the value of K, with larger values resulting in less frequent conflicts as it is less likely that two threads are concurrently operating on the same cluster center. Since threads only occasionally update the same center concurrently, this algorithm benefits from TM's optimistic concurrency

When updating the cluster centers, the size of the transaction is proportional to D, the dimensionality of the space. Thus, the sizes of the transactions in `kmeans` are relatively small and so are its read and write sets. Overall, the majority of execution time for `kmeans` is spent calculating the new cluster centers. During this operation, each thread reads from its partition of objects so no transaction is required. Thus, relatively little of the total execution time is spent in transactions.

5) `labyrinth`: This benchmark implements a variant of Lee's algorithm [26] similar to the *LEE-TM-p-ws* program from [40]. The main data structure is a three-dimensional uniform grid that represents the maze. In the parallel version, each thread grabs a start and end point that it must connect by a path of adjacent maze grid points.

The calculation of the path and its addition to the global maze grid are enclosed by a single transaction. A conflict occurs when two threads pick paths that overlap. To reduce the chance of conflicts, the privatization technique described in [40] is used. Specifically, a per-thread copy of the grid is created and used for the route calculation. Finally, when a thread wants to add a path to the global grid, it revalidates by re-reading all the grid points along the new path. If validation fails, the transaction aborts and the process is repeated, starting with a new, updated copy of the global grid. Transactions are beneficial for implementing this program as deadlock avoidance techniques would be required in a lock-based approach.

Additional performance can be achieved in the program by using early-release [19], as described in [40]. Early-release allows a transaction to remove a data address from its transactional read set so that it does not generate conflicts. However, the programmer or compiler must guarantee that removing the address from the read set does not violate the atomicity of the program. In `labyrinth`, early-release is used to remove all transactional reads generated when copying the global grid to the per-thread copy. As early-release operates with cache line granularity, each maze grid point is padded to occupy an entire cache line to ensure correctness. Finally, since early-release is not available on all TM systems, its use can be disabled when compiling this benchmark.

Overall, almost all of `labyrinth`'s execution time is taken by the path calculation, and this operation also reads and writes an amount of data proportional to the number of total maze grid points. Consequently, `labyrinth` has very long transactions with very large read and write sets. Virtually all of the code is executed transactionally, and the amount of contention is very high because of the large number of transactional accesses to memory.

6) `ssca2`: Scalable Synthetic Compact Applications 2 (SSCA2) [2] is comprised of four kernels that operate on a large, directed, weighted multi-graph. These four graph kernels are commonly used in applications ranging from computational biology to security. For STAMP, we focus on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. This part of the code is well suited for TM as it benefits greatly from optimistic concurrency.

The transactional version of SSCA2 has threads adding nodes to the graph in parallel and uses transactions to protect accesses to the adjacency arrays. Since this operation is relatively small, not much time is spent in transactions. Additionally, the length of the transactions and the sizes of their read and write sets is also small. The amount of contention is the application is

relatively low as the large number of graph nodes leads to infrequent concurrent updates of the same adjacency list.

7) `vacation`: This application implements an online transaction processing system similar in design to SPECjbb2000 [37] but serving the task of emulating a travel reservation system instead of a wholesale company. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates.

Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, `vacation` spends a lot of time in transactions and its transactions are of medium length with moderate read and write set sizes. Low to moderate levels of contention among threads can be created by increasing the fraction of sessions that modify large portions of the database. Finally, using transactions greatly simplified the parallelization as designing an efficient locking strategy for all the data structures in `vacation` is non-trivial.

8) `yada`: The `yada` (Yet Another Delaunay Application) benchmark implements Ruppert's algorithm for Delaunay mesh refinement [32]. The basic data structures are a graph that stores all the mesh triangles, a set that contains the mesh boundary segments, and a task queue that holds the triangles that need to be refined. In each iteration of the algorithm, a skinny triangle is removed from the work queue, its retriangulation is performed on the mesh, and any new skinny triangles that result from the retriangulation are added to the work queue.

The usage of transactions in `yada` is similar to that in [24], but it is applied to a different algorithm in this benchmark. Accesses to the work queue are enclosed by a transaction as is the entire refinement of a skinny triangle. As almost all the execution time is spent calculating the retriangulation of a skinny triangle, this benchmark has relatively long transactions and spends almost all of its execution time in transactions. While performing the retriangulation, several triangles in the mesh are visited and later modified, leading to large read and write sets and a moderate amount of contention. `yada` continuously concurrently modifies a shared graph, which was simpler for us to parallelize with TM than with locks.

C. Configurations and Data Sets

One goal for STAMP is to cover a wide range of transactional execution behaviors in order to stress all aspects of the evaluated TM systems. The differences across the applications discussed above achieve this to some extent: the applications exhibit different transaction lengths, read and write set sizes, percentage of time spent in transactions, amount of contention, and working set size. Transactional characteristics of each STAMP application are summarized qualitatively in Table III.

To provide further coverage, we also exploit the fact that several of the applications exhibit different behavior depending on the size and type of the input data set. Table IV lists each of the STAMP applications and their recommended configurations and data sets. There are six variants of `kmeans` and `vacation` to target different levels of contention and working set sizes. These variants are denoted by appending *-low* and *-high* to the application name to indicate the relative amount of contention. Additionally, the usage of a larger data set is indicated by adding a '+' to the end of the application name. Thus, the six variants of `kmeans` are named `kmeans-high`, `kmeans-high+`, `kmeans-high++` `kmeans-low`, `kmeans-low+`, and `kmeans-low++`. For the remainder of the benchmarks, there are only three variants as increasing the data set size also affects the level of contention. For example, `bayes` has the variants: `bayes`, `bayes+` and, `bayes++`. Finally, the variants without a '++' suffix are intended for running in simulation environments.

TABLE III: Qualitative summary of each STAMP application's runtime transactional characteristics: length of transactions (number of instructions), size of the read and write sets, time spent in transactions, and amount of contention. The description of each characteristic is relative to the other STAMP applications. A quantitative characterization appears later in Section V.

Application	Tx Length	R/W Set	Tx Time	Contention
bayes	Long	Large	High	High
genome	Medium	Medium	High	Low
intruder	Short	Medium	Medium	High
kmeans	Short	Small	Low	Low
labyrinth	Long	Large	High	High
ssca2	Short	Small	Low	Low
vacation	Medium	Medium	High	Low/Medium
yada	Long	Large	High	Medium

D. Implementation

To ease portability of STAMP to several TM systems, we chose to implement all the applications using the C programming language. We also used a low-level API to manually identify parallel threads and insert transaction markers and barriers. The same annotations are used by all TM versions of the code. Moreover, the only difference among the STM and hybrid versions of the code is linkage against a different TM barrier library. Manual optimizations of read and write barriers for accesses to shared data were performed following the guidelines in [1, 18]. Currently, the TM annotations are implemented by C macros, which makes them easy to replace, remove, or port to different systems. Currently, the STAMP applications are also compatible with OpenTM [3], which provides a higher-level representation of the transactions.

IV. Methodology

Two sets of experiments were used to evaluate STAMP. The first set quantitatively verifies the coverage of transactional behaviors by the benchmark suite. The second set demonstrates the portability and practical usefulness of STAMP by running

TABLE IV: Recommended configurations and data sets for STAMP. Suffixes of *-low* and *-high* indicate the relative amount of contention, and appended '+' symbols indicate larger input sizes.

Application	Arguments	Description
bayes bayes+ bayes++	-v32 -r1024 -n2 -p20 -i2 -e2 -v32 -r4096 -n2 -p20 -i2 -e2 -v32 -r4096 -n10 -p40 -i2 -e8 -s1	Dependencies for v variables are learned from r records, which have $n \times p$ parents per variable on average. Edge insertion has a penalty of i, and up to e edges are learned per variable.
genome genome+ genome++	-g256 -s16 -n16384 -g512 -s32 -n32768 -g16384 -s64 -n16777216	Gene segments of s nucleotides are sampled from a gene with g nucleotides. A total of n segments are analyzed to reconstruct the original gene.
intruder intruder+ intruder++	-a10 -l4 -n2048 -s1 -a10 -l16 -n4096 -s1 -a10 -l128 -n262144 -s1	n traffic flows are analyzed, a of which have attacks injected. Each flow has a max of l packets, and the random seed s is used.
kmeans-high kmeans-high+ kmeans-high++ kmeans-low kmeans-low+ kmeans-low++	-m15 -n15 -t0.05 -i random-n2048-d16-c16 -m15 -n15 -t0.05 -i random-n16384-d24-c16 -m15 -n15 -t0.00001 -i random-n65536-d32-c16 -m40 -n40 -t0.05 -i random-n2048-d16-c16 -m40 -n40 -t0.05 -i random-n16384-d24-c16 -m40 -n40 -t0.00001 -i random-n65536-d32-c16	The number of cluster centers used is varied from m to n. A convergence threshold of t is used, and analysis is performed on input i. The input consists of n points of d dimensions generated about c centers.
labyrinth labyrinth+ labyrinth++	-i random-x32-y32-z3-n96 -i random-x48-y48-z3-n64 -i random-x512-y512-z7-n512	The input i consists of a maze of dimensions $x \times y \times z$. n paths are routed.
ssca2 ssca2+ ssca2++	-s13 -i1.0 -u1.0 -l3 -p3 -s14 -i1.0 -u1.0 -l9 -p9 -s20 -i1.0 -u1.0 -l3 -p3	There are 2^s nodes in the graph. The probability of inter-clique edges and unidirectional edges are i and u, respectively. The max path length is l, and the max number of parallel edges is p.
vacation-high vacation-high+ vacation-high++ vacation-low vacation-low+ vacation-low++	-n4 -q60 -u90 -r16384 -t4096 -n4 -q60 -u90 -r1048576 -t4096 -n4 -q60 -u90 -r1048576 -t4194304 -n2 -q90 -u98 -r16384 -t4096 -n2 -q90 -u98 -r1048576 -t4096 -n2 -q90 -u98 -r1048576 -t4194304	The database has r records of each reservation item, and clients perform t sessions. Of these sessions, $u\%$ reserve or cancel items and the remainder create or destroy items. Sessions operate on up to n items and are performed on $q\%$ of the total records.
yada yada+ yada++	-a20 -i 633.2 -a10 -i ttimeu10000.2 -a15 -i ttimeu1000000.2	The input mesh i is refined so that it has a minimum angle of a. The input *633.2* consists of 1264 elements; *ttimeu10000.2*, 19998 elements; and *ttimeu1000000.2*, 1999998 elements.

the suite on several different TM designs and comparing their performance. To ensure a valid comparison between STM and HTM systems, we used an execution-driven simulator for all experiments. Table V presents the main parameters of the simulated multi-core system we used. The processor model assumes an IPC of 1 for all instructions that do not access memory. However, the simulator captures all the memory hierarchy timings including contention and queuing events.

Using the simulator, we ran STAMP on top of the following TM systems:

- **Lazy HTM:** This is an HTM system that follows the TCC architecture [16], except that transactions are only executed for code sections demarcated by transaction boundary markers instead of all code sections. It implements lazy data versioning in caches and performs conflict detection late (when a transaction is ready to commit) by using the coherence protocol. Because the coherence protocol is used for conflict detection, the lazy HTM detects conflicts at cache line granularity. When the lazy HTM overflows the caches' capacity for buffering speculative data, it temporarily serializes the execution of transactions. On conflicts, the lazy HTM restarts the aborted transaction immediately, without any backoff schemes. Note that there is no fundamental reason why HTM cannot implement more

TABLE V: Configuration for the simulated multi-core system.

Feature	Description
Processors	1 to 16 x86 cores, in-order, single-issue
L1 Cache	64KB, private, 4-way, 32B line, 1-cycle access Provides TM bookkeeping for HTM systems
Network	32B bus, split transactions, pipelined, MESI
L2 cache	8MB, shared, 32-way, 32B line, 12-cycle access
Memory	100-cycle off-chip access
Signatures	2048 bits per signature register Provides conflict detection for hybrid TM systems Hash functions: (1) Unpermuted cache line address (2) Cache line address permuted as in [9] (3) Address from (2) shifted right by 10 bits (4) Permuted lower 16 bits of cache line address

sophisticated contention management; not using backoff was just a simple HTM design point we used.

- **Eager HTM:** An HTM system similar to LogTM [29]. It uses the L1 caches and the coherence protocol to implement eager data versioning and early conflict detection, respectively. On conflicts, the requester loses, aborts, and restarts immediately without any backoff. Conflicts are detected at line granularity, and to prevent livelock, transactions are given high priority after aborting 32 times. There is no fundamental reason why eager HTM cannot use more sophisticated contention management (e.g., temporarily stalling transactions), and the previously described policy is simply one we chose. Finally, overflowed addresses are handled by inserting them into a Bloom filter [5]. The Bloom filter participates in the conflict detection mechanism and because of its conservative nature (aliasing of addresses), it may signal false conflicts.
- **Lazy STM:** An x86 port of the TL2 software TM system [13]. It performs lazy versioning using a software write buffer. To provide conflict detection, it uses locks for data in the write set during commit. Conflicts for data in the read set are detected by checking version numbers periodically, and after a transaction aborts three times, the lazy STM uses a randomized linear backoff mechanism. Finally, it detects conflicts at word granularity and provides weak isolation of transactions.
- **Eager STM:** An eager version of TL2. It uses an undo log and holds locks on data in the write set throughout the transaction to provide versioning. Conflict detection is similar to the lazy STM system, and its conflict management, conflict detection granularity, and weak isolation policies are the same.
- **Lazy Hybrid:** A hybrid of the lazy STM that follows the SigTM system [8]. It uses hardware signatures to track the read and write set and to implement fast conflict detection. Data versioning is still in software, and randomized linear backoff is used for contention management. Finally, conflict detection is performed at line granularity and strong isolation of transactions is provided.
- **Eager Hybrid:** The eager equivalent of the lazy hybrid. Conflict detection utilizes hardware signatures but data versioning uses a software undo log. The eager hybrid uses the same contention management policy as the lazy hybrid (and the two STMs) and provides line granularity conflict detection and strong isolation of transactions.

We selected these TM systems as representative points in the TM design space. The conventional wisdom is that HTM systems should perform the best as hardware transparently performs all transactional bookkeeping. Previous studies have shown that HTM systems are up to a factor of four times faster than STM and up to a factor of two faster than hybrid systems [8, 33]. Similarly, it is commonly thought that eager systems should perform better than lazy systems as eager systems perform their memory updates throughout the transaction, instead of waiting till the transaction commit point [29, 34].

STAMP was used to evaluate if the conventional wisdom holds for these designs. A good benchmark suite that stresses a wide range of potential behavior may be able to identify unexpected cases in which the conventional wisdom is wrong. We used the same code (same parallelization strategy and same transaction boundaries) with all systems. The software and hybrid TM systems use the optimized annotations for read and write barriers, but each of them provides different code for the actual barrier functionality. When compiling for HTM systems, the annotations for read and write barriers were ignored (implicit barriers). For the application configurations and data sets, those without the '++' suffix were used as those are the ones designed for use with a simulation environment.

V. Evaluation

In this section, we present a quantitative analysis of the transactional characteristics of STAMP, and use STAMP to compare the performance of six different TM systems.

A. STAMP Basic Characterization

Table VI presents the basic runtime statistics for the STAMP applications and includes data such as the transaction length in instructions, read and write set size in 32-byte cache lines, percentage of execution time spent in transactions, and average number of retries per transaction. All these were measured using the lazy HTM to shield from the implementation details of the barriers necessary for memory accesses to shared variables in STM systems. For the number of read and write barriers and number of STM retries, however, the STMs were used.

All transactional characteristics in Table VI vary at least two orders of magnitude, thus showing that STAMP covers many different transactional execution scenarios. In most of the applications, the transactional statistics follow a normal distribution, but for `genome` and `intruder` they are bimodal and for `vacation` they are trimodal. The modes in these applications are caused by the phased nature of their execution.

The mean number of instructions per transaction ranges from a low of 50 instructions in `bayes` to a high of 687,809 in `labyrinth+`. In `labyrinth`, the large set of operations necessary to find a routing path are packed in a single atomic block. While this coarse-grain approach greatly simplifies code management, it leads to large transactions. In addition to `labyrinth`, `bayes` and `yada` also have long transactions.

The sizes of transactional read and write sets were found by running STAMP on the lazy HTM. The 90^{th} percentile is given as a guide to TM system designers for sizing their hardware transactional buffers to handle the common case. The largest read and write set sizes are found in `labyrinth+`, with values of 783 and 779 cache lines, respectively (24.5 KB and 24.3 KB, respectively). This implies that for HTMs, transactional support at the L2 cache may be needed to avoid the overhead of virtualization mechanisms, especially because of associativity misses. At the other extreme is `ssca2` with a read set of 10 lines and a write set of 4 lines. Since most of the write sets are small, not stalling conflicting transactions in our eager HTM is acceptable since applying the undo log is not too expensive.

TABLE VI: The basic characterization of the STAMP applications. The number of instructions per transaction does not include instructions that occur as part of TM barriers. The lazy HTM was used to measure the read and write sets and the amount of time spent in transactions. The amounts of read and write barriers were collected using the lazy STM, and the lazy HTM was used to find the fraction of time spent in transactions. For the number of retries, 16 threads were used on all systems. The transactional statistics for `genome` and `intruder` follow bimodal distributions, and those for `vacation` are trimodal. The rest of the applications have normal distributions.

Application	Per Transaction					Time in Transactions	Retries Per Transaction				Working Set	
							HTM		STM			
	Instructions (mean)	Read Set (90 pctile)	Write Set (90 pctile)	Read Barrier (90 pctile)	Write Barrier (90 pctile)		Lazy (mean)	Eager (mean)	Lazy (mean)	Eager (mean)	Small (KB)	Large (MB)
bayes	60,584	452	304	24	9	83%	0.66	6.50	0.59	0.66	128	2
bayes+	57,130	448	266	26	9	83%	0.69	5.78	0.61	0.69	128	2
genome	1,717	98	15	32	2	97%	0.10	0.47	0.14	2.20	128	1
genome+	1,709	108	15	30	2	97%	0.02	0.26	0.06	1.14	128	4
intruder	330	51	20	71	16	33%	1.79	6.27	3.54	3.31	32	1
intruder+	331	54	18	54	9	43%	0.67	2.05	1.95	2.96	128	2
kmeans-high	117	14	5	17	17	7%	0.07	0.13	2.73	3.10	16	1
kmeans-high+	153	16	6	25	25	6%	0.05	0.11	3.49	3.68	16	2
kmeans-low	117	14	5	17	17	3%	0.02	0.05	0.89	0.80	16	1
kmeans-low+	153	16	6	25	25	3%	0.01	0.02	0.81	0.70	16	2
labyrinth	219,571	433	458	35	36	100%	0.72	2.64	0.94	1.11	64	1
labyrinth+	687,809	783	779	46	47	100%	2.55	10.59	1.07	1.38	128	2
ssca2	50	10	4	1	2	17%	0.01	0.01	0.00	0.01	256	2
ssca2+	50	10	4	1	2	16%	0.00	0.00	0.00	0.00	512	4
vacation-high	3,223	130	24	432	12	86%	0.37	1.01	0.00	0.01	256	2
vacation-high+	4,193	173	23	608	12	92%	0.25	0.66	0.04	0.05	512	4
vacation-low	2,420	99	22	287	8	86%	0.07	0.25	0.00	0.00	256	2
vacation-low+	3,161	126	22	401	8	92%	0.05	0.18	0.02	0.03	512	4
yada	9,795	250	142	256	108	100%	0.52	3.06	2.51	4.35	32	2
yada+	11,596	274	145	282	108	100%	0.45	2.04	1.38	2.52	64	8

The numbers of read and write barriers were measured with the lazy STM and are identical for the eager STM and hybrids. The number of read barriers varies from a low of 1 in `ssca2` to a high of 608 in `vacation-high+`. In comparison, the amounts of write barriers range from 2 in `genome` and `ssca2` to 108 in `yada`. Overall, the number of read barriers is typically much larger than the number of write barriers, suggesting that designers of STMs should especially optimize read barriers.

Over 80% of the execution time is spent in transactions by five of the eight applications: `bayes`, `genome`, `labyrinth`, `vacation`, and `yada`. These applications use transactions frequently as their algorithms continuously operate on shared data structures. Such applications put great stress on the underlying TM system as any inefficiencies will be amplified by the frequent use of transactions. STAMP also covers applications that use transactions sporadically. Three of the benchmarks (`intruder`, `kmeans`, and `ssca2`) spend less than half of their execution time in transactions, as there are significant portions of the code that operate on easily-identified private data. Over all the STAMP applications, the time spent in transactions ranges from a low of 3% in `kmeans-low` to a high of 100% in `labyrinth` and `yada`.

The last transactional characteristic measured in Table VI is the average number of times a transaction retries before successfully committing. This statistic was measured by using 16 threads on the six TM systems, and it is highly dependent on both the number of threads and the underlying TM system. Moreover, the actual amount of work lost is a function of both the number of retries and the point in the transaction at which the retry occurs. Nevertheless, the data for the number of retries gives quantitative evidence that the STAMP applications cover a wide spectrum of contention cases ranging from virtually no retries ($<$0.01 for `ssca2+`) to 10.59 retries for `labyrinth+`. Finally, good contention management policies should be able to reduce the number of retries in most of the TM systems, and the numbers for this transactional characteristic show that STAMP can be used to evaluate contention management policies as well.

Finally, the last two columns of Table VI, indicate two of the working sets exhibited by each of the programs. These values were found by setting the cache size to all power-of-2 sizes from 16 KB to 64 MB and looking for points at which significant changes in the miss rate occurred. Like the other columns in the table, these values cover a variety of different scenarios. Applications in which the small working set exceeds the capacity of the L1 cache (e.g., `ssca2` and `vacation`) are likely to have a significant portion of the execution time spent on cache misses.

B. Performance Analysis for TM Systems

We measured the performance of 20 variants of the STAMP applications on the six TM systems as we varied the number of cores from 1 to 16. The speedup curves (normalized to sequential execution with code that does not have extra overhead

from the annotations for threads, transactions, or barriers) are shown in Figure 1.

1) bayes: This application has the interesting result that the relative performance among the TM systems is the *opposite* of the expected ranking. As shown in Table VI, bayes has relatively large read and write sets. Consequently, both of the HTMs experience overflows and suffer large performance hits. In particular, the lazy HTM handles overflows by temporarily serializing execution of transactions and the eager HTM handles overflows by using a Bloom filter to summarize the overflowed addresses. Because the Bloom filter is conservative in nature, it can cause the eager HTM to abort transactions more than necessary (but never less). Transaction aborts are especially expensive in eager data versioning TM systems as they have to process the undo logs to rollback a transaction.

Another difference among the TM systems is the granularity at which they detect conflicts. Because the HTMs and the hybrids use the cache coherence protocol to also perform TM conflict detection, they are limited to detection at line granularity. On the other hand, the STMs are free of this restriction and use word granularity instead. Conflict detection at finer granularity turns out to be particularly important in this case. The HTMs and hybrid TMs suffer from a large number of false conflicts.

As a result, the eager HTM performs the worst among all the TM systems and the lazy HTM does not perform much better. In contrast, because the STMs implement data versioning in software, they have almost no overflow-related performance penalties and perform much better than the HTMs. As for the hybrid TMs, their performance is between that of the HTMs and STMs. Finally, one more thing to note about bayes is that the execution time is sensitive to the order in which dependencies are learned. Thus, the speedup curves are not as smooth as those for other STAMP applications.

2) genome: Early conflict detection turns out to be disadvantageous in genome. For example, the eager STM has an average number of retries per transaction almost 20× that of the lazy STM. This conflict detection policy causes the eager STM and eager hybrid to both experience a high number of aborts and suffer from livelock. In comparison, the eager HTM's contention management policy prevents it from falling prey to this pathology. In particular, when a transaction aborts several times in the eager HTM, the eager HTM system promotes the transaction's priority level and prevents any other transactions from aborting it. This technique allows the eager HTM to continue making forward progress and results in its good performance on genome.

Overall, the HTMs are about twice as fast as the lazy STM, and the eager STM fails to scale because its eager conflict detection results in livelock. However, among the three eager TM systems, the relative performance is as expected, as is that among the lazy TM systems. Finally, the performances of the eager STM and eager hybrid will likely improve with contention management that addresses livelock problems.

3) intruder: For the TM systems we implemented, one advantage that the STMs and hybrids have over the HTMs arises under workloads with high contention. TMs with a software component can have more sophisticated contention management policies, and in particular, the STMs and hybrids use a randomized linear backoff scheme after a transaction aborts. Additionally, the overhead caused by resetting the transaction state in software (e.g., processing the software undo logs) also serves as a backoff mechanism. In contrast, the two HTM implementations do not use a backoff scheme and simply restart the transaction as soon as possible. Not having a sophisticated contention management policy is not intrinsic to HTM, and we just chose to have a simple design.

Among the STAMP applications, intruder has a relatively high amount of contention. Thus, whereas the performance of the STMs and hybrids follow expected behavior, the HTMs do not perform as anticipated. The effect is more pronounced in the eager HTM, because aborts are more expensive in an eager scheme (time to apply the undo log). The early conflict detection in eager HTM also results in livelock. Hence, for this application, there is an HTM system (eager HTM) that performs worse than the STMs, and neither of the HTMs scale beyond 8 cores. Overall, the lazy HTM performs the best, and with more sophisticated contention management, the HTMs will likely scale better on this benchmark.

4) kmeans: In this benchmark, all six TM systems scale similarly and their relative performance follows the expected behavior; however, the actual speedups differ greatly. Because kmeans uses transactions infrequently and has relatively short transactions with few conflicts, the performance difference between the corresponding eager and lazy variants is very small for the HTM and hybrid designs. For the STMs, however, the eager variant is noticeably faster than the lazy one because the eager STM has shorter read barriers (the read barrier in the lazy STM must first search in the write buffer to check if the data have been previously written within the same transaction). Overall, the HTMs perform 2–4× better than the STMs, and the performance of the hybrids is about halfway between the HTMs and STMs.

5) labyrinth: In this benchmark, early-release is used to remove the reads generated by the grid copying operation from the transactional read set. The usage of early-release is necessary for the HTMs as all memory accesses are transparently tracked by the hardware (i.e., implicit barriers). For the STMs and hybrids, however, the TM system relies on the annotation of accesses to shared objects with read and write barriers. Thus, by not using any read barriers to perform the grid copying, the STMs and hybrids avoid the need for early-release. As a result, the number of read and write barriers is only proportional to the length of the routed path instead of to the total number of grid points. Therefore, since there are relatively few TM barriers, all the TM systems perform similarly.

Performance differences among the TM systems occur in labyrinth+ because of the larger data set. As noted in Section V-A, the transactional read and write sets for labyrinth+ are comparable to the size of the L1 cache. This causes both HTMs to overflow and pay performance penalties.

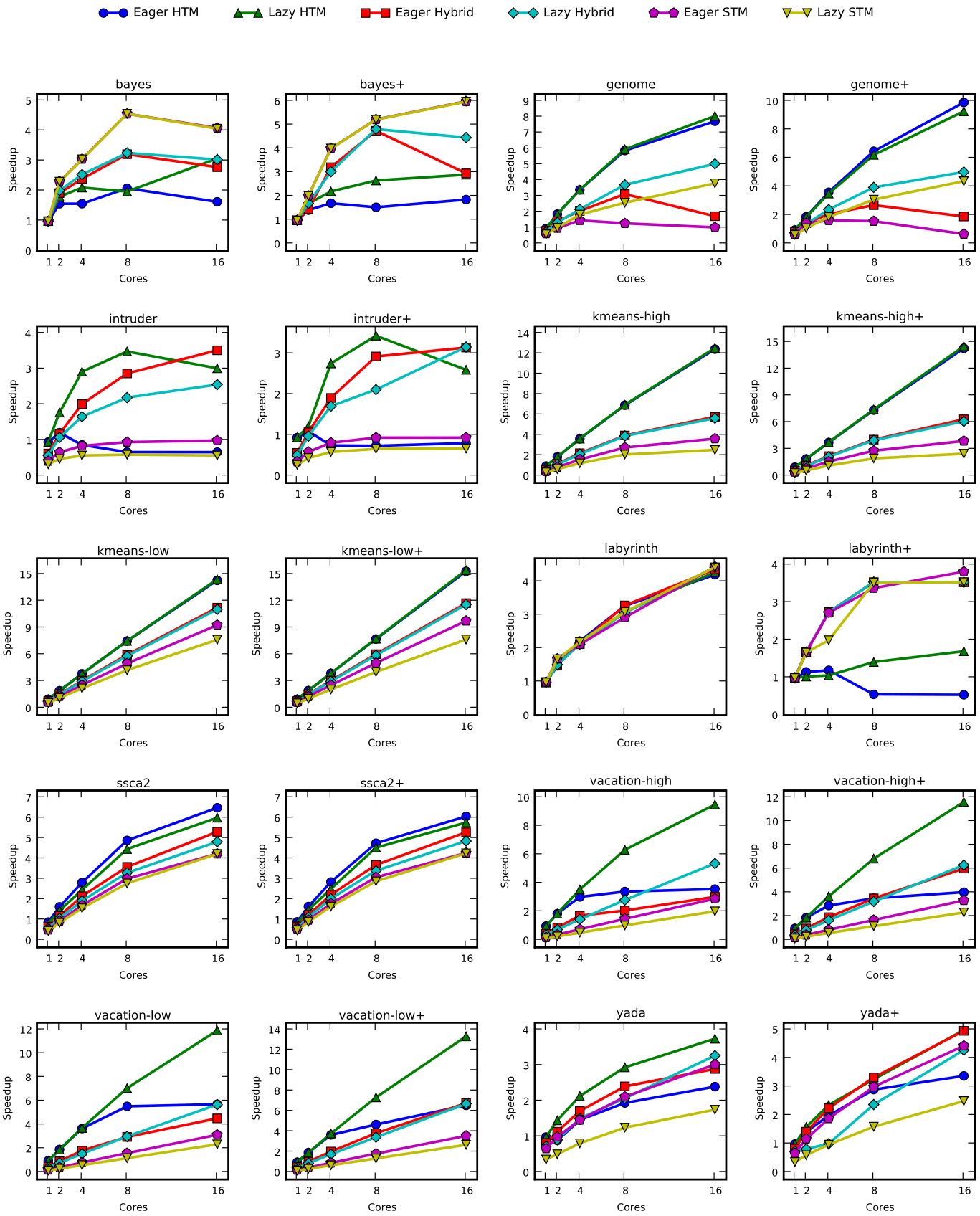

Fig. 1: Speedups over sequential code on each of the six TM implementations. Application variants that use the larger data set are indicated by a '+' appended to the application name. The suffixes *-low* and *-high* indicate variants with low and with high contention, respectively.

Moreover, recall that the eager HTM handles overflows by recording overflowed addresses in a Bloom filter. To ensure correctness, the eager HTM cannot perform early-release on addresses that hit in the Bloom filter. This causes the eager HTM to perform worse than the lazy HTM as its transactions abort more frequently. In contrast to the HTMs, the STMs and hybrids do not experience overflow-related performance problems as the data versioning is handled in software. Thus, their performance is about the same as with the small data set.

6) ssca2: Like kmeans, ssca2 has very short transactions with very small read and write sets, and less than 20% of the execution time is spent in transactions. Consequently, all TM systems perform well and operate with minimal overhead. Overall, the relative ranking among the six TM systems is as expected.

7) vacation: All the TM systems follow the expected behavior in vacation except for the eager HTM and eager hybrid. As in bayes, conflict detection at a finer granularity is particularly advantageous, and Table VI shows that the STMs' word-level granularity conflict detection leads to much fewer aborts than the HTMs' line-level granularity conflict detection. However, for the STMs, the performance advantage gained by finer granularity conflict detection is outweighed by the overhead caused by the large number of read barriers used to provide concurrent accesses to the database trees.

Finally, for the HTMs and hybrids, lazy data versioning is more advantageous than eager data versioning. In the latter scheme, transaction aborts are more expensive as an eager data versioning TM system must apply undo logs to rollback a transaction. Thus, when coupled with the moderately long transactions in vacation, the eager HTM performs much worse than the lazy HTM and the same pattern is seen between the two hybrids. At 16 cores in vacation-high, the large amount of work lost to violations in the eager HTM even causes its performance to fall below that of the lazy hybrid.

8) yada: In yada, transactions have relatively large read and write sets, which cause the HTMs to suffer from overflows. In the lazy HTM, overflows cause temporary serialization of transactions, and in the eager HTM, overflows cause more frequent transaction aborts. On the other hand, the large numbers of read and write barriers do not cause the STMs to incur a significant performance penalty.

As a result, in yada the lazy HTM performs the best and is closely followed by the hybrids and eager STM. Because transaction aborts are expensive for TMs with eager versioning (time to process the undo log), the eager HTM is only able to outperform the lazy STM. With the larger data set in yada+, however, overflows are more expensive because transactions are longer. This causes the lazy HTM to serialize transactions for longer periods of time and the eager HTM to lose more work to transaction aborts. Consequently, the performance of the hybrids and STMs relative to the HTMs improves.

C. Performance Summary

In general, the applications in STAMP perform well with all six TM systems. In most cases, there are significant speedups and the relative differences among HTM, hybrid, and STM systems were as expected. However, there were a few cases where the conventional wisdom did not hold. In bayes and labyrinth+, for example, the performance of the HTMs suffered because the large number of transactional reads generated overflows, causing either transaction execution to become serialized (lazy HTM) or excessive aborts to occur (eager HTM). In bayes, the finer granularity in conflict detection allowed STMs to unexpectedly outperform both the HTMs and the hybrids. Comparing the HTM systems, the results indicate that the two schemes (eager and lazy) either perform similarly (e.g., kmeans and genome), or the lazy HTM system performed better as it guarantees forward progress in high contention scenarios (e.g., intruder and vacation-high).

None of the shortcomings of the six TM systems identified above are necessarily fundamental. Future research may be able to address them and remove the resulting performance loss. Our point is that STAMP's coverage of a diverse set of scenarios in transactional execution allows us to stress each TM system thoroughly and identify its (sometimes unexpected) shortcomings. Hence, STAMP can be a valuable tool for future research on TM systems.

VI. Conclusions

In creating STAMP, we sought to create the first benchmark suite for TM that adequately addressed *breadth* in application domains and algorithms well suited for programming with transactions, *depth* in stressing a wide range of transactional execution cases, and *portability* across a wide range of TM systems. To demonstrate these properties we ran 20 variants of the eight STAMP applications on six different TM systems. All the measured transactional characteristics ranged at least two orders of magnitude, and although the TM systems' relative performance was often the expected behavior, STAMP helped identify cases that contradict conventional wisdom and require further research.

The source code for STAMP is freely available from http://stamp.stanford.edu. It is our hope that STAMP will help address a great need in the TM research community by providing a comprehensive tool to help people design and evaluate TM systems. Already there are many early adopters of STAMP in both industry and academia [7, 10]. Future work for STAMP includes the addition of more benchmarks to cover even more scenarios.

VII. Acknowledgments

We would like to thank the anonymous reviewers and early users of STAMP for their feedback at various stages of this work. We also want to thank Sun Microsystems for making the TL2 code available and IBM Research for their advice on SSCA2. This work was supported by NSF grant number 0720905 and 0546060. Chí Cao Minh is supported by a Stanford Graduate Fellowship.

REFERENCES

[1] A.-R. Adl-Tabatabai, B. Lewis, et al. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 2006.

[2] D. A. Bader and K. Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC '05: 12th International Conference on High Performance Computing*. December 2005.

[3] W. Baek, C. Cao Minh, et al. The OpenTM transactional application programming interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pp. 376–387. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2944-5.

[4] C. Bienia, S. Kumar, et al. The parsec benchmark suite: Characterization and architectural implications. Tech. Rep. TR-811-08,, Princeton University, 2008.

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 1970.

[6] C. Blundell, J. Devietti, et al. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pp. 24–34. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-706-3.

[7] J. Bobba, N. Goyal, et al. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[8] C. Cao Minh, M. Trautmann, et al. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.

[9] L. Ceze, J. Tuck, et al. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, pp. 227–238. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2608-X.

[10] S. Cherem, T. Chilimbi, et al. Inferring locks for atomic sections. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pp. 304–315. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-860-2.

[11] D. M. Chickering, D. Heckerman, et al. A Bayesian approach to learning Bayesian networks with local structure. In *UAI '97: Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pp. 80–89.

[12] P. Damron, A. Fedorova, et al. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. October 2006.

[13] D. Dice, O. Shalev, et al. Transactional locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*. March 2006.

[14] R. Guerraoui, M. Kapalka, et al. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007. ISSN 0163-5980.

[15] B. Haagdorens, T. Vermeiren, et al. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *WISA '04: Proceedings of the 5th International Workshop on Infomation Security Applications*, pp. 188–203. 2004.

[16] L. Hammond, V. Wong, et al. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pp. 102–113. June 2004.

[17] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 388–402. ACM Press, 2003. ISBN 1-58113-712-5.

[18] T. Harris, M. Plesko, et al. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, USA, 2006.

[19] M. Herlihy, V. Luchangco, et al. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pp. 92–101. ACM Press, New York, NY, USA, July 2003. ISBN 1-58113-708-7.

[20] M. 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*, pp. 289–300. 1993.

[21] A. Jaleel, M. Mattina, et al. Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pp. 88–98, 11-15 Feb. 2006. ISSN 1530-0897.

[22] H. Jin, M. Frumkin, et al. The openmp implementation of nas parallel benchmarks and its performance. Tech. rep.

[23] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. ISSN 0018-8646.

[24] M. Kulkarni, K. Pingali, et al. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 211–222. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-633-2.

[25] S. Kumar, M. Chu, et al. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, New York, NY, USA, March 2006.

[26] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, Sep. 1961.

[27] V. J. Marathe, M. F. Spear, et al. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

[28] A. W. Moore and M. S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[29] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *12th International Conference on High-Performance Computer Architecture*. February 2006.

[30] R. Narayanan, B. Ozisikyilmaz, et al. Minebench: A benchmark suite for data mining workloads. *Workload Characterization, 2006 IEEE International Symposium on*, pp. 182–188, Oct. 2006.

[31] C. Perfumo, N. Sonmez, et al. Dissecting transactional executions in haskell. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. August 2007.

[32] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[33] B. Saha, A. Adl-Tabatabai, et al. Architectural support for software transactional memory. In *MICRO '06: Proceedings of the International Symposium on Microarchitecture*. 2006.

[34] B. Saha, A.-R. Adl-Tabatabai, et al. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, New York, NY, USA, March 2006.

[35] M. L. Scott, M. F. Spear, et al. Transactions and privatization in delaunay triangulation. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pp. 336–337. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-616-5.

[36] A. Shriraman, M. F. Spear, et al. An integrated hardware-software approach to flexible transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*. June 2007.

[37] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. http://www.spec.org/jbb2000/, 2000.

[38] Standard Performance Evaluation Corporation, *SPEC OpenMP Benchmark Suite.* http://www.spec.org/omp.

[39] E. Vallejo, T. Harris, et al. Hybrid transactional memory to accelerate safe lock-based transactions. In *TRANSACT '08: Third ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. February 2008.

[40] I. Watson, C. Kirkham, et al. A study of a transactional parallel routing algorithm. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 388–398. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2944-5.

[41] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36. June 1995.