RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems

Gokcen Kestor*
Barcelona Supercomputing Center
gokcen.kestor@bsc.es

Vasileios Karakostas
Barcelona Supercomputing Center
vasilis.karakostas@bsc.es

Osman S. Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Adrian Cristal
IIIA - Artificial Intelligence Research Institute CSIC - Spanish National Research Council
adrian.cristal@bsc.es

Ibrahim Hur
Barcelona Supercomputing Center
ibrahim.hur@bsc.es

Mateo Valero†
Universitat Politècnica de Catalunya
mateo@ac.upc.es

ABSTRACT

Transactional Memory (TM) has been proposed as an alternative concurrency mechanism for the shared memory parallel programming model. Its main goal is to make parallel programming for Chip Multiprocessors (CMPs) easier than using the traditional lock synchronization constructs, without compromising the performance and the scalability. This topic has received substantial research attention and several TM designs have been proposed using various TM benchmarks. We believe that the evaluation of TM proposals would be more solid if it included realistic applications, that address on-going TM research issues, and that provide the potential for straightforward comparison against locks.

In this paper, we introduce *RMS-TM*, a Transactional Memory benchmark suite composed of seven real-world applications from the Recognition, Mining and Synthesis (RMS) domain. In addition to featuring current TM research issues such as nesting and I/O and system calls inside transactions, the RMS-TM applications also provide a mix of short and long transactions with small/large read and write sets with low/medium/high contention rates. These characteristics, as well as providing lock-based versions of the applications, make RMS-TM a useful TM tool. Current TM benchmarks do not explore all these features. In our evaluation with selected STM and HTM systems, we find that our benchmark suite is also scalable, which is useful for evaluating TM designs on high core counts.

*G. Kestor is Ph.D. student at Universitat Politècnica de Catalunya, Barcelona, Spain

†M. Valero is the director of Barcelona Supercomputing Center, Barcelona, Spain

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming ; D.0 [**Software**]: [benchmark suite]

General Terms

Performance, Measurement, Experimentation

Keywords

transactional memory, benchmark suite, performance measurement

1. INTRODUCTION

While innovations in process technology increase the number of transistors on a die, the performance gains from more complex cores and larger caches diminish. Therefore, chips with multiple cores have quickly become a de-facto standard. Multi-core systems have the potential for significant performance improvements, but the complexity of parallel programming and the difficulty of writing efficient and correct code limit the effective use of these systems.

New programming models have been proposed to ease the development of parallel applications that perform well on multi-core architectures. Transactional Memory (TM) [16] is one of such programming models that enables programmers to perform multiple memory operations atomically without worrying about the complexity issues associated with other programming models such as locks. However, it is unclear whether TM implementations provide sufficient performance when compared to locks, in particular fine-grained locking.

Although multiple Software TM (STM) [15, 21, 30, 32] and Hardware TM (HTM) implementations [8, 14, 22, 34] have been proposed, there are still open research issues, in addition to performance, such as handling nested transactions, I/O operations, system and library calls inside transactions. Moreover, performance comparison of TM-based applications against their equivalent lock-based versions is crucial for the justification of further research in this area as well as for convincing the industry to implement TM systems in commercial products. One major aspect of perform-

ing functional and performance evaluation of TM systems is the development of a good TM benchmark suite.

We identify six desired properties for a TM benchmark suite: (1) the suite should include both the lock-based and TM-based versions of the same benchmarks, (2) the benchmarks should have good scalability, (3) the benchmarks should represent real-world applications, (4) the benchmarks should encompass a wide range of different TM behaviors, (5) the benchmarks should include open research issues for TM researchers, and (6) the benchmark suite should be useful in evaluating both STM and HTM systems.

Although there are multiple benchmark suites [2, 7, 13, 35, 38] proposed for evaluating TM systems, none of those has all of the above-mentioned properties. For example, the *STAMP* benchmark suite [7] does not include lock-based versions of its applications, *SPLASH-2* [35] does not provide a wide range of TM characteristics, *Atomic Quake* [38] cannot be used to evaluate HTM systems. Previous work by Hughes et al. [17] also pointed out that existing TM workloads have similar characteristics in terms of transactional behaviours and that there is need of more comprehensive benchmarks. In this paper, we introduce such a benchmark suite, *RMS-TM*. Apart from having a wide range of transactional and run-time characteristics, RMS-TM presents challenging features such as nested transactions, I/O operations and library calls inside transactions, which are common operations in real applications.

To construct our benchmark suite, we develop a step by step methodology for choosing candidate TM benchmarks from among a set of real-world applications, and we reimplement the selected applications by using the TM programming model. The final result is a new benchmark suite that includes different applications from the Recognition, Mining, and Synthesis (RMS) domain. We use RMS applications because these applications have high relevance to mainstream workloads, and they are proposed as good workloads to evaluate future multi- and many-core systems [19].

This paper makes the following contributions:

- We introduce a new benchmark suite, RMS-TM, that consists of lock-based and transactified versions of seven applications from *BioBench* [1],*MineBench* [25], and *PARSEC* [4] benchmark suites. RMS-TM has a wide range of transactional and run-time characteristics that qualify it as a new and comprehensive benchmark suite for evaluating both STM/HTM designs. The applications in our benchmark suite feature the following: 1) representative real-world applications, 2) nested transactions [23, 26], 3) large amount of I/O operations [3], system [33] and library calls inside atomic blocks, 4) complex function calls and control flow inside atomic blocks, 5) various mix of long/short transactions with different sizes of read/write sets, 6) low/medium/high contention rates, and 7) good scalability.

- We develop a methodical procedure to construct our benchmark suite from candidate applications. We first divide the application selection process into *static* and *dynamic pre-transactification* phases, and then, in the *transactification* phase, we transactify the selected applications from their original lock-based parallel implementations. This process ensures that the selected applications satisfy the desired properties for a TM benchmark suite.

- We evaluate our benchmark suite using three different TM implementations (one STM and two HTMs), namely *Intel-STM* [32], *EazyHTM* [34], and *Scalable TCC* [8] and we show that RMS-TM can be used in the evaluation of both STM and HTM systems.

  We find that the RMS-TM applications present varying percentage (1.5%-95.7%) of time spent inside atomic blocks with small and large read (a few bytes to 3 MB) and write (a few bytes to 493 KB) sets, and with low and high contention (0.0%-88.4% abort rates). We also find that our benchmarks have good scalability (Intel STM 4.7×, EazyHTM 6.0×, and ScalableTCC 6.3×, on average, for eight threads).

The rest of this paper is organized as follows. The next section summarizes prior work. Section 3 covers our methodology for selecting and transactifying representative TM benchmarks. In Section 4 we describe the applications in our benchmark suite, and we evaluate them in Section 5. Finally, we provide concluding remarks in Section 6.

2. RELATED WORK

In this section, we briefly review some of the previously proposed TM benchmarks to highlight their advantages and disadvantages in evaluating TM systems. We categorize TM benchmark suites into micro-benchmarks (*TM micro-benchmarks* [11] and *The Haskell STM Benchmark Suite* [28]), parallel benchmarks (*SPLASH-2* [35]), and other benchmarks with more complex transactional characteristics (*STMBench7* [13], *Lee-TM* [2], *WormBench* [37], *STAMP* [7], *Atomic Quake* [38]. and *QuakeTM* [12].

TM micro-benchmarks contain single data structures, such as hash tables, linked lists, B-trees, etc. These benchmarks are useful for providing basic-level insights into TM designs, but they do not exhibit different TM characteristics, and they are not representative of realistic workloads.

The Haskell STM Benchmark Suite consists of ten applications that are implemented with Haskell, which features TM as a first-class language feature. Most of the applications in this benchmark suite are micro-benchmarks.

SPLASH-2 contains eight parallel applications and four computational kernels. This benchmark suite focuses on applications that utilize little synchronization between threads, and it does not provide various sizes of critical sections or different conflict rates. Therefore, this benchmark suite is not fully capable of evaluating the underlying TM system and discovering interesting transactional behaviors.

STMBench7 presents an application to analyze STM systems. This benchmark provides a coarse-grained and medium-grained locking implementation that can be compared to its equivalent transactified version. The benchmark performs complex and dynamic operations on a large data structure, so it has only relatively long transactions.

Lee-TM benchmarks feature long and realistic workloads that consist of sequential as well as coarse- and medium-grained lock-based, transactional, and optimized transactional implementations. This benchmark suite is useful for comparing different lock and transactional implementations; however, it only features different implementations of the same algorithm.

WormBench is a highly configurable transactional application. This synthetic application is useful mostly to mimic

Table 1: Applications that pass the Static Pre-Transactification step.

Application	Domain	Locking Type	Nested Locking	Function Calls	Special Operations in Critical Sections	Barrier Synchronization
Hmmsearch	sequence profile searching	coarse-grained	no	yes	I/O, memory management operations, library calls	no
Hmmpfam	sequence profile searching	coarse-grained	no	yes	I/O, memory management operations, library calls	no
Hmmcalibrate	calibrate profile HMMs	coarse-grained	no	yes	memory management operations, library calls	no
Apriori	association rule mining	coarse-grained fine-grained	yes	yes	memory management operations	yes
PLSA	dynamic programming	fine-grained	no	yes	none	no
Rsearch	pattern recognition mining	fine-grained	no	yes	memory management operations	no no
ScalParC	classification	coarse-grained fine-grained	no	no	none	yes
UtilityMine	association rule mining	coarse-grained fine-grained	yes	yes	memory management operations	yes
Bodytrack	computer vision	fine-grained	no	yes	library calls	yes
Fluidanimate	fluid simulation	fine-grained	no	no	none	yes
Freqmine	frequent item set mining	fine-grained	no	no	memory management operations	no no

existing TM applications rather than discovering unknown usage patterns of emerging transactional applications.

STAMP is a benchmark suite that consists of eight applications with 30 different sets of configurations. The input data for the applications present a wide range of runtime transactional characteristics, e.g., varying transaction lengths, read/write set sizes, and degree of contention. This benchmark suite provides sequential and transactional versions of the applications, but it does not provide their lock-based versions; thus, TM researchers cannot compare TM-based and the equivalent lock-based implementations.

QuakeTM and *Atomic Quake* are rich and complex transactional memory applications. QuakeTM is parallelized from the sequential version of Quake game server using TM, while Atomic Quake is derived from the parallel lock-based version of the server. These benchmarks exhibit irregular parallelism, have I/O and system calls, error handling, and instances of privatization. In addition, inside transactions, there are function calls, memory management, and nested transactions. However, these benchmarks can only be used for evaluating STM systems due to their size and complexity.

In comparison, RMS-TM includes lock-based and TM-based implementations of seven real-world applications that have a wide range of TM characteristics in terms of transaction lengths, read/write set sizes, and contention. This benchmark suite is suitable for evaluating both STM and HTM systems. In addition, unlike most other TM benchmarks, RMS-TM presents many desirable properties, such as nested transactions, I/O operations, and system calls inside transactions.

3. THE TRANSACTIFICATION PROCESS

In this section we describe our methodology for constructing the RMS-TM benchmark suite. To create our benchmark suite, we develop a two-step procedure: (1) we apply static and dynamic pre-transactification to select applications from among a set of candidate benchmarks, and (2) we transactify the selected applications.

We analyze three different benchmark suites: BioBench, MineBench, and PARSEC. The applications in these benchmark suites are from the RMS domain, and they represent future workloads [19]. The BioBench suite consists of bioinformatics applications that are developed using the `Pthread` parallel programming model [6]. The MineBench suite is designed considering data mining categories that are commonly used in industry problems. The applications in this suite are implemented by using `OpenMP` [9]. The PARSEC benchmark suite includes emerging applications that are computationally intensive.

3.1 Pre-Transactification Phase

We choose applications from the candidate benchmark suites using TM-specific usefulness criteria, e.g., having nested transactions, irrevocable operations, system and library calls inside atomic blocks, etc. To make an effective and comprehensive analysis, we divide the pre-transactification phase into two sub-phases: static and dynamic. In the static phase, we analyze source codes of the applications; in the dynamic phase, we execute and profile the candidate applications to calculate the amount of time they spend inside critical sections and to analyze their scalability.

3.1.1 Static Pre-Transactification

We use five criteria in the static pre-transactification phase: (1) synchronization constructs used between lock blocks, (2) type of locking granularity, (3) nested locking, (4) function calls between acquiring and releasing locks, and (5) special operations inside critical sections, e.g., I/O operations, library and system calls.

Table 1 shows the characteristics of the applications selected in the static pre-transactification phase. We select *Hmmpfam*, *Hmmsearch*, and *Hmmcalibrate*, because they exhibit a large amount of I/O operations, system and library calls, and relatively complex function calls inside critical sections. *Hmmpfam* and *Hmmsearch* also present a large number of instructions in coarse-grained critical sections. Applications that have a coarse-grained locking structure are good candidates, because they spend a significant amount of time waiting to acquire a lock; minimizing this synchronization time is an important topic for TM research.

ScalParC, *Apriori*, and *UtilityMine* include both fine- and coarse-grained locking, providing different types and sizes of transactions. In addition, they use synchronization constructs between atomic blocks. We expect placement of synchronization constructs between lock blocks to create inter-

Table 2: Time spent (%) inside critical sections for the lock-based applications. The data sets used are appended to the application name.

Application	Number of Threads			
	1	2	4	8
Hmmsearch	0.3	0.4	0.4	0.5
Hmmpfam	11.1	12.0	14.2	20.6
Hmmcalibrate	3.9	4.2	4.8	5.6
Apriori-100	1.1	1.6	2.8	5.6
Apriori-1000-20	0.1	0.2	0.4	0.7
Apriori-2000-20	0.1	0.1	0.2	0.4
PLSA	0.0	0.0	0.0	0.0
Rsearch	0.0	0.0	0.0	0.0
ScalParC-A64-D125	0.0	0.2	1.0	1.9
ScalParC-A64-D250	0.0	0.1	0.6	0.8
ScalParC-A64-500	0.0	0.1	0.4	0.7
UtilityMine-1000-10-1	53.9	52.8	56.8	56.3
UtilityMine-1000-10-20	70.1	66.0	70.0	69.5
UtilityMine-2000-20-1	69.8	65.6	69.5	65.7
Fluidanimate	0.0	5.5	9.6	15.2
Freqmine	0.0	0.0	0.0	0.0
BodyTrack	0.1	0.2	0.3	0.2

esting TM characteristics, e.g., a high abort rate even when an application does not spend much time inside transactions. In fact, immediately after a barrier, all threads will attempt to enter their atomic blocks at the same time, but only one will commit successfully.

PLSA, *Rsearch*, *BodyTrack*, *Fluidanimate*, and *Freqmine* pass the static pre-transactification phase as well as. Since these applications have function calls inside critical sections, it is difficult to statically determine the length of the transactions and their read/write sets. In addition, some of these applications have memory management operations inside critical sections.

3.1.2 Dynamic Pre-Transactification

In the dynamic pre-transactification phase, we use percentage of time spent inside critical sections and scalability as the evaluation criteria.

Table 2 shows that *PLSA*, *Rsearch*, *Freqmine*, and *BodyTrack* spend a very small percentage of their execution time inside critical sections. These applications cannot stress the underlying TM systems due to their short transaction lengths, low transaction frequencies, and small read/write sets; therefore, we filter out these applications. Even though *ScalParc* and *Apriori* spend a short amount of time inside critical sections, we maintained these applications in the benchmark suite because they have several marked atomic blocks and they use synchronization constructs, e.g., barriers between consecutive atomic blocks. *Apriori* and *UtilityMine* have a high level (up to nine) of nested locking, which makes them good candidates for evaluating TM systems with support for arbitrary levels of nested transactions. From the *Hmmer* package, we select *Hmmsearch*, *Hmmpfam*, and *Hmmcalibrate*. Although *Hmmsearch* spends a short time inside critical sections, it is a crucial benchmark for TM research because it has I/O operations and library and system calls inside critical sections.

Figure 1 shows the scalability of lock-based applications that we consider as good candidates for TM research. Notice that all the benchmarks have a sub-linear speedup but they scale well except when we use eight threads in parallel, i.e., all the avaliable processors in our experimental setup.

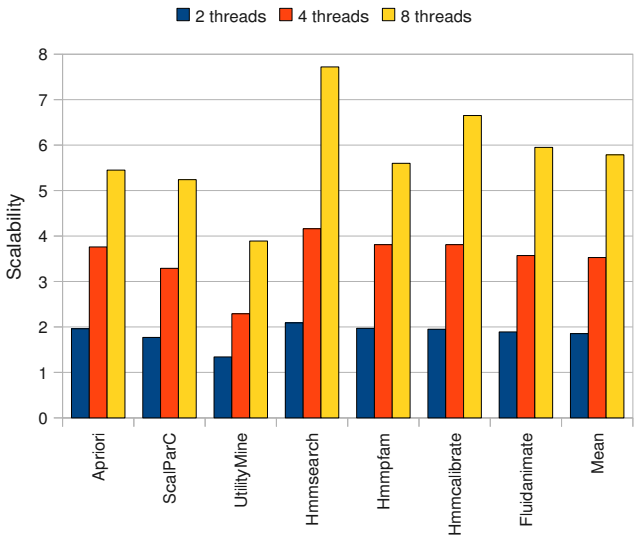


Figure 1: Scalability of the lock-based applications, with the largest data sets, normalized to single-threaded lock execution time with Intel STM.

3.2 Transactification Phase

We transactify the selected applications starting from their equivalent lock-based versions by replacing locks with transactions. To maintain the original semantics, we keep the size of the atomic blocks as in the lock-based versions.

The transactification process is not straightforward because each application has a different parallelization strategy. Moreover, each TM system poses specific challenges, e.g., calls to pre-compiled library functions and I/O operations and system calls inside transactions. We now describe the details of these challenges and our solutions for three TM systems, namely Intel STM, EazyHTM, and ScalableTCC.

3.2.1 STM Implementation

Intel STM [32] consists of a C/C++ compiler and a high-performance STM Runtime Library. The compiler instruments all shared memory reads and writes inside transactions by using read and write barriers. The flattening model is used to support nested transactions, and weak isolation between transactional and non-transactional code is provided. Transactions can be executed in optimistic or pessimistic mode. In both cases, the transactional writes update the data in-place with strict two-phase locking, while the transactional reads are executed optimistically or pessimistically. Serial execution mode is also provided to support transactions that contain irrevocable operations.

Intel STM compiler provides simple language extensions to develop TM applications. The functions inside atomic blocks should be marked as either `tm_callable`[1] or `tm_pure`[2]. Otherwise, if an unannotated function is called inside atomic blocks, the compiler generates code that triggers serial ex-

[1]The compiler generates a clone function annotated as `tm_callable` and translates each memory `read` and `write` to a TM read barrier function and a TM write barrier function.
[2]The programmer guarantees that a function marked as `tm_pure` does not access shared variables when it is called from inside a transaction.

ecution unless it knows that the called function does not require instrumentation. The applications that we examine often allocate objects through the `new` operator and/or they call external functions inside atomic blocks. The version of the compiler that we use[3] does not mark the `new` operator as `tm_callable` implicitly although the object constructor is marked. This causes transactions to run irrevocably. To deal with this challenge we overload the `new` operator and we mark it as `tm_callable`. Another challenge is associated with function calls of precompiled libraries inside transactions. To avoid executing these transactions in serial mode, we reimplement some glibc string functions, such as `strcmp`, `strstr`, `strlen`, and we mark them as `tm_callable`.

3.2.2 HTM Implementations

EazyHTM [34] and ScalableTCC [8] are recent HTM proposals that provide scalable performance. Both TM systems are directory-based and implement lazy data versioning. The key feature of EazyHTM is separating conflict detection and conflict resolution. Conflicts are detected while transactions run, but they are resolved at commit time allowing truly parallel commits. On the other hand, ScalableTCC detects conflicts optimistically when transactions are ready to commit. ScalableTCC implements a continuous use of transactions within parallel programs providing non-blocking execution and improved fault-isolation.

The main challenges that we faced while porting RMS-TM applications to EazyHTM and ScalableTCC, are dynamic memory management and I/O operations inside transactions. Most of our applications dynamically allocate memory using `malloc` and `realloc`. To overcome this issue, we use a user mode memory manager that allocates chunks of memory for each thread when the applications start [7]. When a thread requires new memory, the user mode manager takes this memory from its pre-allocated pool and assigns it to the thread without calling the `malloc` system call.

In addition, *Hmmpfam* and *Hmmsearch* perform many I/O operations inside critical sections. The replacement of the locks protecting these critical sections with transactions is not straightforward because rollback can happen at any time during the execution of a transaction, and the transaction can restart at any arbitrary point of its execution. Most current TM systems cannot safely perform I/O or system calls inside transactions. For these operations, we use the library developed by Perfumo et al. [29], which enables the use of I/O operations inside transactions. To provide a fair comparison, we also modify the lock-based versions of the applications to make them use the same library.

4. RMS-TM OVERVIEW

We used our pre-transactification process to select applications from the RMS domain, and we transactified those applications to construct the RMS-TM benchmark suite. In this section, we provide the descriptions of the applications in the benchmark suite: *Hmmsearch*, *Hmmpfam*, and *Hmmcalibrate* from BioBench, *Apriori*, *ScalParC*, and *UtilityMine* from MineBench, and *Fluidanimate* from PARSEC.

TM-Hmmsearch reads an HMM and searches a sequence database for significantly similar sequence matches. In the transactional version, the threads read the next sequence from an input list of sequences in parallel, and they use transactions to protect the accesses to the input list of sequences. Moreover, the threads share two score lists ranked by per-sequence scores and per-domain scores and a histogram of the whole sequence stores. Transactions are used to protect update operations on these data structures.

TM-Hmmpfam searches a query sequence against a profile HMM database. In the transactional version, each thread accesses the shared profile HMMs database and reads the next profile HMM. This application scores the input sequence against the profile HMM and adds a significant hit to the per-sequence and per-domain top hits lists. Transactions protect the shared file pointer of the HMMs database. Update operations on the shared per-sequence and per-domain top hits lists are also enclosed inside transactions.

TM-Hmmcalibrate calibrates a profile HMM using an artificial database of sequences. After reading the profile HMM, this application generates random sequences; it computes a raw score for each sequence against the profile HMM and it adds this score to a histogram. The increment on the shared counter and the generation of the sequence are enclosed in transactions. Another transaction is used to protect the accesses to the histogram of scores.

TM-Apriori [36] is an Association Rule Mining (ARM) algorithm performed on transactional records in a database. This application uses a hash tree to store candidates. Transactions are used to protect the calculation of support values and the insertion of a candidate item set into the hash tree.

TM-ScalParC [18] is a parallel formulation of a decision tree classification. The decision tree model splits the records in the training set into subsets based on the values of attributes. This process continues until each record entirely consists of examples from one class. During the partitioning phase, different threads try to simultaneously access a shared counter. Transactions protect the accesses to this shared counter.

TM-UtilityMine [20] is another ARM technique. A utility mining model is developed to identify item sets with high utilities. The utility of an item or an item set can be defined as its usefulness. A single common hash tree stores the candidate item sets at each level of search as well as their transaction-weight utilization. Transactions protect the updates of the utility of item sets and insertion of a candidate into the tree.

TM-Fluidanimate [24] is based on spatial partitioning and uses a uniform grid partitioned to cells to reside fluids. The uniform grid is evenly partitioned in subgrids along cell boundaries. We use transactions to enclose the update particles of the cells that lie on subgrid boundaries.

5. EVALUATION

We evaluate RMS-TM using three different (one STM and two HTMs) TM systems, namely Intel STM, EazyHTM, and ScalableTCC. We compare the TM-based implementations of the applications to their equivalent lock-based versions and we analyze their transactional behavior, such as read/write set sizes, abort/commit rates, time spent inside atomic blocks, scalability, etc. We also evaluate the STAMP benchmark suite on the same TM systems and we compare and contrast the results with our benchmark suite.

5.1 Intel STM Results

We now present the evaluation of our benchmark suite using the Intel STM system. All results are the averages of

[3]Intel C++ STM Compiler Prototype Edition 3.0

Table 3: Basic TM characteristics (for eight threads) of the RMS-TM applications, with Intel STM. The number of bytes read/written transactionally and the number of aborts or commits are generated by the Intel STM runtime library.

Application	Read Set (bytes)			Write Set (bytes)			Transactions		
	Min	Mean	Max	Min	Mean	Max	#Commits	#Aborts	Abort Rate (%)
TM-Hmmsearch	24	3K	3M	0	296	493K	613,316	7,678	1.2
TM-Hmmpfam	16	7K	2M	0	846	270K	28,333	5,832	17.1
TM-Hmmcalibrate	8	13K	74K	4	5K	30K	10,016	76,219	88.4
TM-Apriori-100	4	424	67K	0	274	45K	14,410	282	1.9
TM-Apriori-1000-20	4	408	132K	0	263	87K	14,431	290	2.0
TM-Apriori-2000-20	4	449	380K	0	289	246K	14,758	464	3.0
TM-ScalParc-A64-D125	8	31	952	1	7	238	52,404	61,072	53.8
TM-ScalParc-A64-D250	8	30	840	1	7	210	75,408	80,691	51.7
TM-ScalParc-A64-D500	8	34	944	1	8	236	117,240	153,872	56.8
TM-UtilityMine-1000-10-1	32	424	28K	4	7	202	43,724,391	292,031	0.7
TM-UtilityMine-1000-10-20	4	646	65K	4	7	1K	197,213,249	1,212,087	0.6
TM-UtilityMine-2000-20-1	4	644	47K	0	7	1K	3,954,033,044	2,181,138	1.0
TM-Fluidanimate	4	8	1K	4	7	12	1,177,944,500	252	0.0

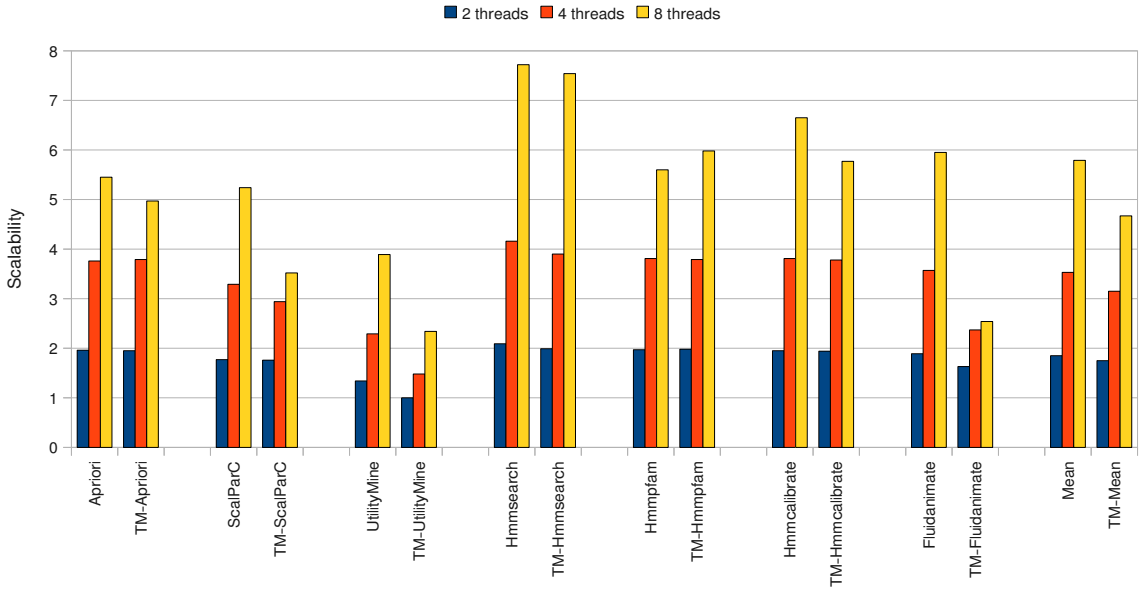


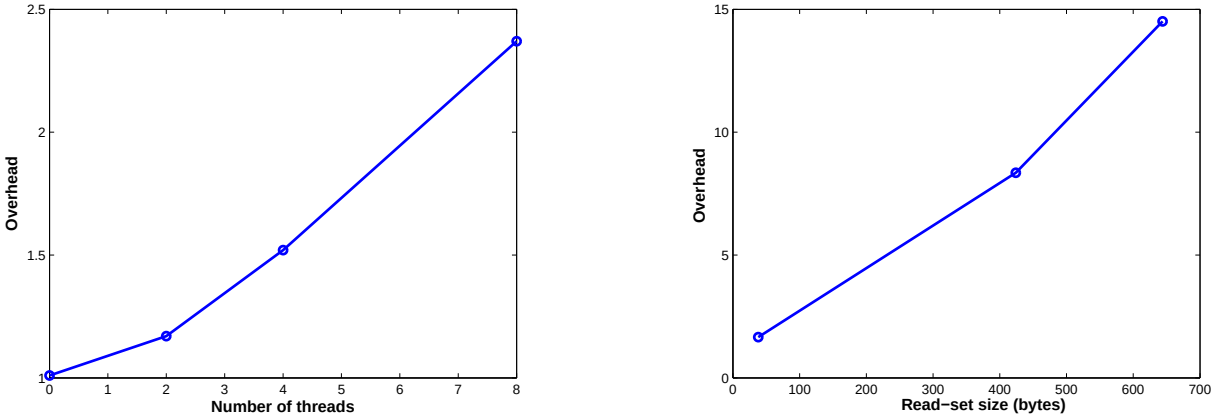
Figure 2: Scalability of the lock-based and TM-based applications, with the largest data sets, normalized to single-threaded lock and TM execution time, respectively, with Intel STM.

five different executions using three different data sets. We perform our experiments on a Dell PE6850 workstation with 4 dual core x64 Intel Xeon processors running at 3.2GHz equipped with 32GB RAM, a 32KB IL1 and a 32KB DL1 private caches per core, a 4MB L2 cache shared by two cores, and a 8MB L3 cache shared by all cores.

5.1.1 Transactional Behavior

Table 3 presents the basic runtime TM characteristics of the RMS-TM applications, such as the number of bytes read or written transactionally, the number of times a transaction retries execution due to a conflict, etc. RMS-TM explores several combinations of TM characteristics: medium read/write sets with medium abort rates (*TM-Hmmpfam*), small read/write sets with high abort rates (*TM-ScalParC*), and large read/write sets with high abort rates (*TM-Hmm calibrate*). In addition, the data for abort rates in Table 3 show that the RMS-TM applications cover a wide spectrum of contention ranging from 0.0% for *TM-Fluidanimate* to 88.4% for *TM-Hmmcalibrate*. Although *TM-ScalParC* spends most of its execution time outside atomic blocks, it has a high abort rate due to the use of synchronization points between consecutive atomic blocks, which confirms our observation in the static pre-transactification phase.

Table 4 presents the percentage of time spent in atomic blocks with respect to total parallel time with 1, 2, 4, and 8 threads for each data set. We observe some overhead introduced by the Intel STM compiler and run-time library because of the extra work required to handle transactions, such as when detecting conflicts. As we can see from Table 2 and Table 4, the Intel STM runtime introduces different overheads in the transactified versions of the benchmarks. For example, the lock version and TM version of *TM-Hmmpfam* spend 20.6% and 20.7% of their parallel times inside critical sections. On the other hand, *TM-ScalParC-A64-D250* spends 0.8% of its parallel time inside critical sections with the lock implementation and 7.4% with the TM implementation. Between these two extremes of the spectrum, there are



(a) TM-Fluidanimate

(b) TM-UtilityMine

Figure 3: (a) Runtime overhead of *TM-Fluidanimate* as a function of the number of threads (from 1 to 8) with constant read- and write-set sizes. (b) Run-time overhead of *TM-UtilityMine* increases as the size of the read-set increases.

Table 4: Time spent (%) inside atomic blocks for RMS-TM applications.

Application	Number of Threads			
	1	2	4	8
TM-Hmmsearch	1.1	1.1	1.2	1.6
TM-Hmmpfam	11.1	12.0	14.2	20.7
TM-Hmmcalibrate	7.8	8.3	9.3	14.3
TM-Apriori-100	3.4	5.1	9.7	17.2
TM-Apriori-1000-20	0.0	0.2	0.6	1.8
TM-Apriori-2000-20	0.2	0.3	0.7	1.5
TM-ScalParC-A64-D125	0.1	0.5	2.3	11.5
TM-ScalParC-A64-D250	0.1	0.3	1.5	7.4
TM-ScalParC-A64-D500	0.1	0.2	1.0	5.9
TM-UtilityMine-1000-10-1	88.7	91.8	91.5	92.2
TM-UtilityMine-1000-10-20	95.3	96.0	95.4	95.6
TM-UtilityMine-2000-20-1	95.2	95.8	95.4	95.7
TM-Fluidanimate	0.0	18.9	39.3	61.7

intermediate cases. For example, *Hmmsearch* spends 0.5% in the lock-based version and 1.6% in the transactified version. Table 4 shows that the benchmarks cover a wide range of cases in terms of time spent inside atomic blocks. This variety is a desirable property for a TM benchmark suite, because it allows researchers to evaluate TM systems using applications that are either very sensitive to TM overheads (*TM-ScalParC-A64-D250*) or those that are not sensitive to the overhead of TM systems (*TM-Hmmpfam*).

5.1.2 Performance Analysis

Figure 2 shows the scalability of the RMS-TM applications with respect to their single-threaded case. The RMS-TM applications present a scalability similar to their equivalent lock-based versions except *TM-Scalparc*, *TM-UtilityMine*, and *TM-Fluidanimate*. Several factors may influence the scalability of TM applications, but a high abort rate is the most common reason for poor scalability. Table 3 shows that *TM-ScalParC* exhibits this characteristic with 56.8% percent abort rate which causes performance degradation especially with eight threads. Although *TM-UtilityMine* has a low abort rate, this benchmark presents a large number of transactions, each one with large read/write sets. In other words, each rollback operation is expensive (the cost of each rollback depends on the read/write set size) and it affects performance. We also found that *TM-UtilityMine*, with 8 threads, spends 66.0% of its total time inside transactions for rollback operations (wasted work) [28].

We performed a deeper analysis of all the applications using `oprofile` [27] and we examined specific performance counters. We found that the Intel STM run-time system evicts data from the L2 cache while managing the read- and write-sets. This increases the number of L2 cache misses and degrades performance. *TM-UtilityMine* is sensitive to this situation: because of its long transactions with large read sets, more than 90% of the L2 cache misses are caused by the Intel STM library. This extra overhead becomes larger as the sizes of the read- and write-sets increase, therefore, it limits the scalability of the application. Nevertheless, *TM-UtilityMine* enables TM designers to have better understanding of the runtime overhead of TM systems.

Scalability is also affected by the run-time STM library. Every time a thread attempts to modify a memory location inside a transaction, the STM run-time system scans the read-set of each active transaction to check whether the same memory location was previously read by another thread. The larger the read-set the longer the time required to scan each active transaction and the larger the overhead introduced by the STM run-time system, which limits scalability. On the other side, the larger the number of concurrent active transactions (which is upper bound by the number of threads), the larger the overhead. Figure 3(a) shows the runtime overhead of *TM-Fluidanimate* as function of the number of threads (from 1 to 8) with constant read- and write-set sizes. The run-time overhead linearly increases with respect to the number of threads.[4] In addition, Figure 3(b) demonstrates that the run-time overhead of *TM-UtilityMine* increases with the size of the read-set.[5] Note that applications with high abort rates will interrupt

[4]For this application the number of transactions per thread is constant.

[5]The number of threads (eight) is constant in this graph.

Table 5: Transactional behavior of the RMS-TM applications with eight threads, with EazyHTM. The sizes of transactional read and write sets are presented as the 90th percentile.

Application	Read Set (cache lines)		Write Set (cache lines)		Transactions		
	90 pctile	Max	90 pctile	Max	#Commits	#Aborts	Abort Rate (%)
TM-Hmmsearch	161	975	56	1,368	2,008	362	15.3
TM-Hmmpfam	3,348	10,338	1,400	3,832	308	345	52.9
TM-Hmmcalibrate	51	71	29	37	5,016	376	7.0
TM-Apriori-100	11	40	6	206	11,232	36	0.3
TM-ScalParC-A64-D125	4	4	3	3	50,393	18,979	27.4
TM-UtilityMine-1000-10-1	65	120	1	2	43,724,391	374,050	0.8
TM-Fluidanimate	2	2	1	1	9,347,885	3,131	0.0

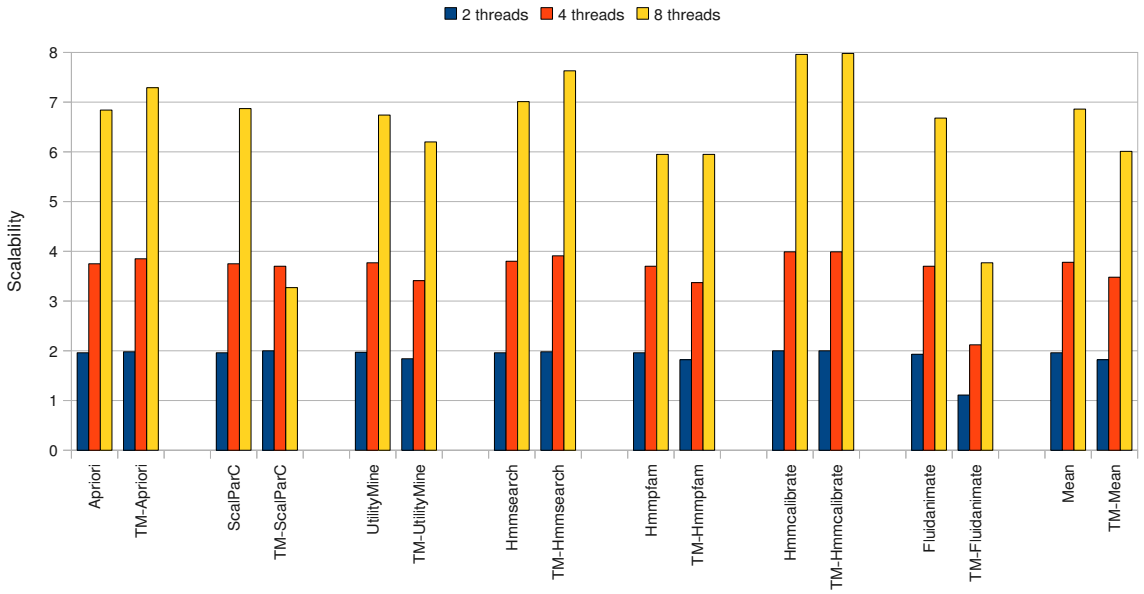


Figure 4: Scalability of the lock-based and TM-Based applications normalized to single-threaded lock and TM execution time, respectively, using EazyHTM. The datasets are indicated in Table 5.

Table 6: Configuration of the simulated system.

Feature	Description
CPU	1-8 Alpha cores, 2 GHz, in-order, 1 IPC
L1	32 KB, 64-byte cache line, 4-way associativity, private per core, writeback, MSI, 2 cycles latency
L2	512 KB, 64-byte cache line, 8-way associativity, private per core, writeback, 8 cycles latency
Main Memory	100 cycles latency
ICN	2D Mesh topology, 10 cycles latency per hop

the list traversal sooner because of conflict detection, and they will have a lower performance degradation. Obviously, applications that spend a large part of their execution inside transactions are affected more by the STM run-time overhead.

5.2 EazyHTM Results

We now evaluate the performance of RMS-TM applications on EazyHTM [34] using a full-system simulator based on the Alpha 21264 architecture. EazyHTM is implemented using the M5 simulator [5] which is modified with a directory memory hierarchy and a core-to-core interconnection network. Table 6 presents the main characteristics of the simulated system. We use the largest possible data set in our simulations.

5.2.1 *Transactional Behavior*

Table 5 summarizes the transactional characteristics of the RMS-TM applications on EazyHTM. *TM-Hmmpfam* exhibits a high abort rate. This is caused by the large read/write sets that do not fit in the cache. Since EazyHTM does not provide support for unbounded transactions, transactions are eventually aborted and restarted. Benchmarks with high commit rates (*TM-Hmmcalibrate*, *TM-Apriori*, *TM-UtilityMine*, and *TM-Fluidanimate*) and with high abort rates (*TM-Hmmsearch*, *TM-Hmmpfam*, and *TM-ScalParC*) are good candidates to evaluate both lazy and eager data versioning. For example, Hammond et al. [14] and Moore et al. [22] show that high commit/abort rates have large impacts on performance in HTM systems. This happens because eager data versioning relies on the idea that the commit rate is higher than the abort rate, therefore, these systems are designed with a low commit cost. On the other hand, HTM systems with lazy data versioning do not rely on this hypothesis and they usually show that the abort cost is significantly lower than the commit cost. To enable researchers to perform exhaustive studies on TM systems

Table 7: Transactional behavior of the RMS-TM applications with eight threads, with ScalableTCC. The sizes of transactional read and write sets are presented as the 90th percentile.

Application	Read Set (cache lines)		Write Set (cache lines)		Transactions			Wasted
	90 pctile	Max	90 pctile	Max	#Commits	#Aborts	Abort Rate (%)	(%)
TM-Hmmsearch	109	945	56	1,369	2,008	204	9.2	0.2
TM-Hmmpfam	3,260	10,342	1,312	3,833	308	219	41.6	8.4
TM-Hmmcalibrate	47	72	26	37	5,016	285	5.3	0.1
TM-Apriori-100	11	19	6	103	14,438	14	0.1	0.6
TM-ScalParC-A64-D125	4	5	3	4	50,352	10,010	16.6	12.9
TM-UtilityMine-1000-10-1	65	120	1	3	43,724,391	436,698	1.0	1.7
TM-Fluidanimate	2	2	1	1	9,347,885	2,207	0.0	0.1

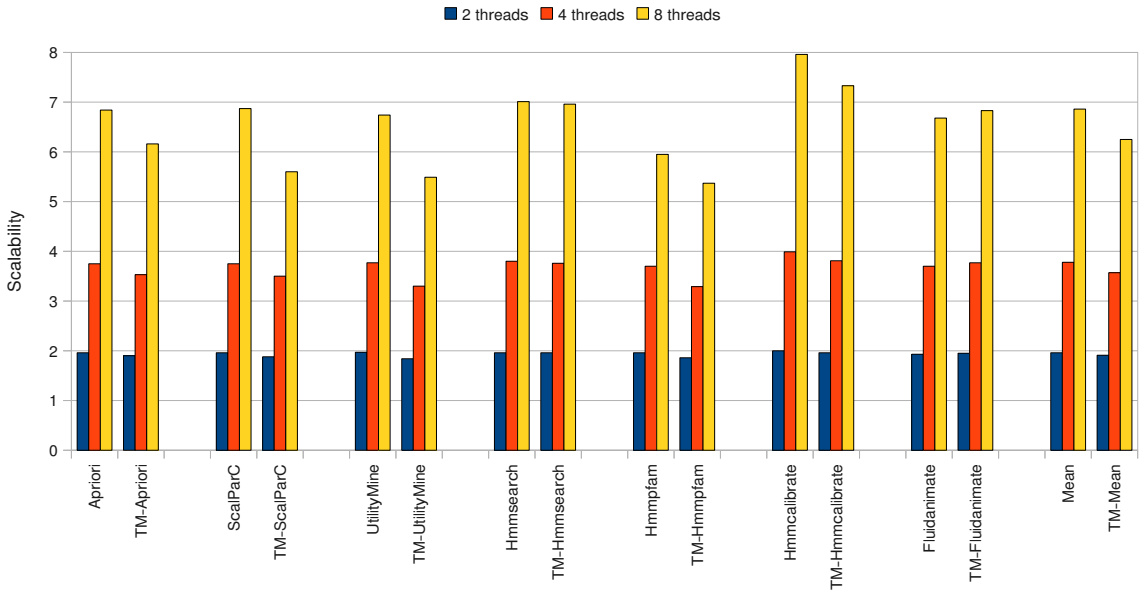


Figure 5: Scalability of the lock-based and TM-based applications normalized to single-threaded lock and TM execution time, respectively, with ScalableTCC. The datasets are indicated in Table 7.

with different versioning strategies, RMS-TM provides different combinations of TM behaviors.

5.2.2 *Performance Analysis*

Figures 4 shows the scalability of lock- and TM-based RMS-TM applications on EazyHTM. The majority of the TM-based applications exhibit good scalability, comparable to their equivalent lock-based versions. More in details, *TM-Hmmcalibrate* scales linearly, *TM-Hmmsearch* and *TM-Apriori* scale slightly better than their lock-based version, while *TM-UtilityMine* scales slightly worse than its lock-based version. *TM-ScalParC* shows a good scalability for up to four threads. However, this application scales poorly with eight threads as opposed to its lock-based version. We noticed that the number of directory messages to detect conflicts is constant with two and four threads (where the application shows a good scalability) but it doubles with 8 threads (the case of poor scalability). *TM-ScalParC* is the only application with such behavior. *TM-Fluidanimate* presents a very high number of directory messages that increases with the number of threads. For all the other applications the number of directory messages is roughly constant regardless of the number of threads. We conclude that EazyHTM's conflict detection mechanism introduces overhead that limits the scalability of *TM-ScalParC* with eight threads and *TM-Fluidanimate* with two, four and eight threads.

5.3 ScalableTCC Results

In this section, we present our experimental results for the ScalableTCC HTM system using a full-system simulator based on the Alpha 21264 architecture. Table 6 presents the main parameters of the simulated multi-core system that we use for ScalableTCC.

5.3.1 *Transactional Behavior*

Table 7 presents the basic TM characteristics of the RMS-TM applications, and it includes data such as the number of commits/aborts and read/write set size in 64-byte cache lines. All transactional characteristics in Table 7 show that RMS-TM covers different combinations of TM execution scenarios, such as the sizes of transactional read (2 - 3,260 cache lines) and write (1 - 1,312 cache lines), and abort rates (0.0% to 41.6%). More specifically, *TM-Hmmpfam* has the largest read- and write-sets, 3,260 (203 KB) and 1,312 (82 KB) cache lines, respectively. Moreover, this application presents the highest abort rate (41.6%). Effective contention manager policies can reduce the number of aborted transactions, which implies that *TM-Hmmpfam* can enable TM

designers to improve their contention manager proposals. On the other hand, *TM-UtilityMine* and *TM-Fluidanimate* show high commit rates with a large number of committed transactions, which makes them desirable TM benchmarks for evaluating TM systems with lazy data versioning, where the commit cost is high.

Figures 5 shows the scalability of lock- and TM-based RMS-TM applications on ScalableTCC. Most of the TM-based applications present similar scalability to their equivalent lock-based versions except *TM-ScalParC*, *TM-Apriori*, *TM-UtilityMine* and *TM-Hmmpfam*. As we can see from Table 7, *TM-ScalParC* and *TM-Hmmpfam* with eight threads waste 12.9%, and 8.4% of their total execution time, respectively, which limits their scalability. Further analysis showed that rolling back aborted transactions is a large component of the total wasted time for these applications. *TM-Apriori* and *TM-UtilityMine* do not scale as well as their lock-based equivalent with eight threads. For these applications, we observe that they spend relatively large amount of time at synchronization points especially with eight threads, as opposed to the other applications.

5.4 Comparison of RMS-TM and STAMP

In this section we compare RMS-TM to STAMP using three different TM systems. Both RMS-TM and STAMP have substantial number of applications with varying abort /commit rates and small/large transactions. RMS-TM also has I/O operations, library calls, memory management operations, pre-compiled library calls inside transactions, and nested transactions, whereas STAMP only provides memory management operations inside transactions. Hence, we believe that the RMS-TM applications present more realistic use cases of TM. On the other hand, the STAMP benchmarks provide larger read/write sets than RMS-TM. This characteristic can help TM researchers evaluate their TM proposals that support virtualized transactions [10].

We analyze the scalability of RMS-TM and STAMP applications on three different TM systems. Figures 6(a), 6(b) and 6(c), show that RMS-TM applications scale well as the number of cores increases on both STM and HTMs (Intel STM 4.7×, EazyHTM 6.0×, and ScalableTCC 6.3×, on average, with eight threads). However, some STAMP applications on the STM implementation, Figure 6(a), show no scalability regardless of the number of threads, whereas they have a reasonable scalability on HTMs (EazyHTM 4.1× and ScalableTCC 3.7×, on average, with eight threads).

Unlike STAMP, RMS-TM provides both lock-based and transactified implementations to better understand drawbacks of TM proposals through direct performance comparison. For example, as presented in Section 5.1.2, performance and scalability analysis between *TM-UtilityMine* and its equivalent lock-based implementation provide important insights into the STM system. With this information we understand that *TM-UtilityMine's* poor scalability is caused by STM run-time overhead rather than the algorithm. Finally, RMS-TM consists of applications written in C/C++ programming languages using different parallel programming models such as `OpenMP` and `Pthread`. On the other hand, STAMP applications are implemented in C with `pthread`.

6. CONCLUSIONS

We introduced a new TM benchmark suite, RMS-TM, that consists of multi-core workloads from the Recognition, Mining, and Synthesis domain. We developed a general methodology to determine applications that are suitable for analyzing TM implementations, and we transactified the selected applications. Therefore, RMS-TM includes both locked-based and transactified versions of the same applications. We evaluated RMS-TM using one STM and two HTM implementations, and we presented the detailed analysis of our experimental results. We found that the applications in our benchmark suite have good scalability, and they feature a wide range of transactional characteristics. RMS-TM is publicly available in the hopes of helping researchers to design and evaluate their TM systems [31].

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their useful comments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation and by the European Commission FP7 project VELOX (216852). Gokcen Kestor is also supported by a scholarship from the Government of Catalonia. Vasileios Karakostas is also partially supported by an Erasmus Grant.

8. REFERENCES

[1] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, Washington, DC, USA, 2005.

[2] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Agia Napa, Cyprus, 2008.

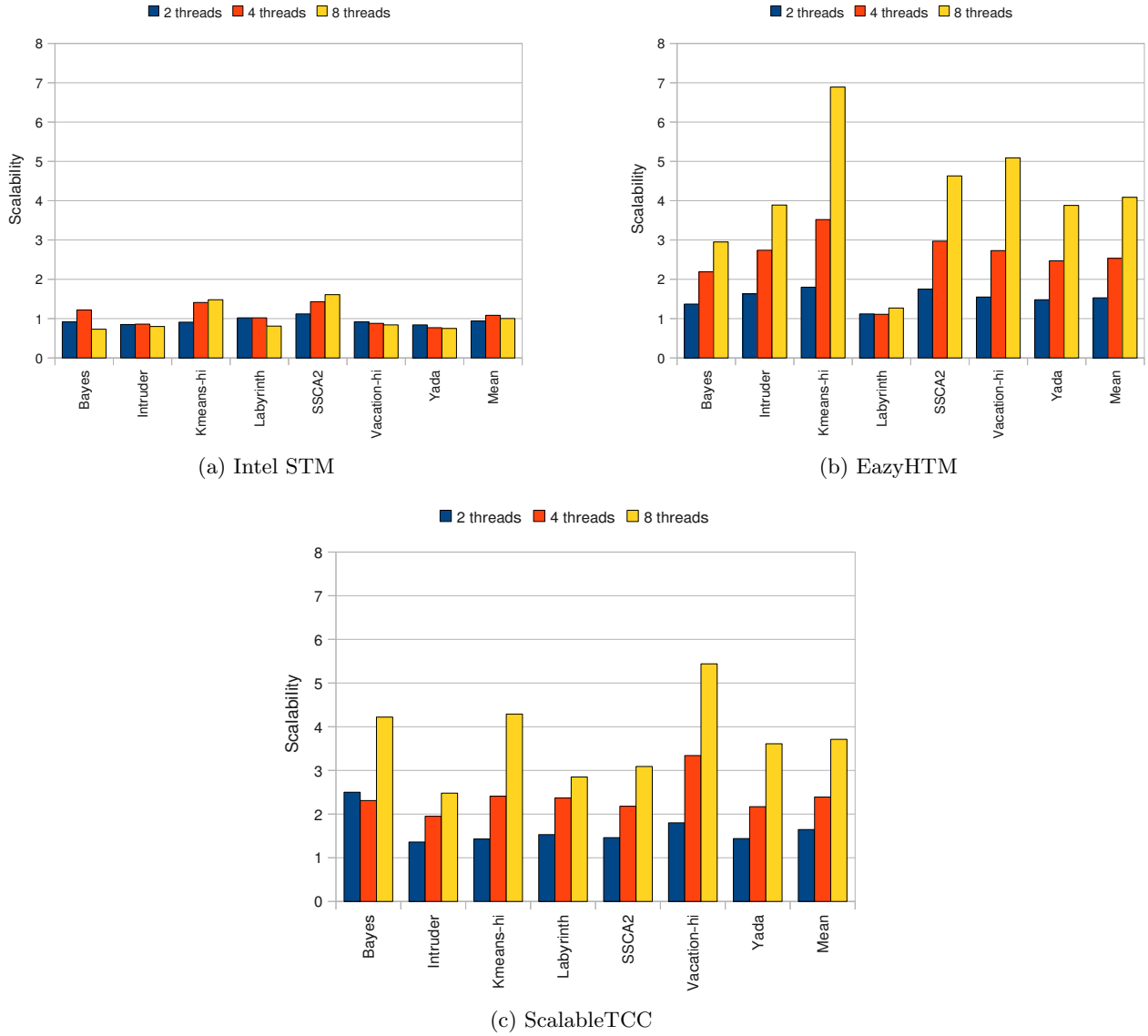
[3] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, pages 54–62, Washington, DC, USA, 2008.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, ON, Canada, 2008.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] C. Cao 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, USA, 2008.



(a) Intel STM

(b) EazyHTM

(c) ScalableTCC

Figure 6: Scalability of the STAMP applications normalized to single-threaded TM execution time.

[8] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 97–108, Washington, DC, USA, 2007.

[9] B. Chapman, G. Jost, and R. Van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *SIGPLAN Notices*, 41(11):347–358, 2006.

[11] J. Ennals, R. *Adaptive Evaluation of Non-Strict Programs. PhD thesis*. University of Cambridge, 2004.

[12] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd Int. conference on Supercomputing*, ICS, 2009.

[13] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. *SIGOPS Operating Systems Review*, 41(3), 2007.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, New York, NY, USA, 2004.

[15] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional

memory. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications*, pages 253–262, Portland, OR, USA, 2006.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993.

[17] C. Hughes, J. Poe, A. Qouneh, and T. Li. On the (dis)similarity of transactional memory workloads. In *2009 IEEE International Symposium on Workload Characterization, IISWC 2009*, 2009.

[18] M. V. Joshi, G. Karypis, and V. Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium*, pages 573–579, Washington, DC, USA, 1998.

[19] B. Liang and P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. In *Technology@Intel Magazine*, pages 1–10, 2005.

[20] Y. Liu, W. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *Proceedings of the 1st International Workshop on Utility-based Data Mining*, pages 90–99, Chicago, IL, USA, 2005.

[21] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of software transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2006*. Ottawa, ON, Canada, 2006.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, Austin, TX, USA, 2006.

[23] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, San Jose, CA, USA, 2006.

[24] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, San Diego, CA, USA, 2003.

[25] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the International Symposium on Workload Characterization*, pages 182–188, San Jose, CA, USA, 2006.

[26] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, San Jose, CA, USA, 2007.

[27] OProfile - a system profiler for linux. Available at http://oprofile. sourceforge.net/.

[28] C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 67–78, Ischia, Italy, 2008.

[29] C. Perfumo, O. Unsal, A. Cristal, and M. Valero. TxFS: Transactional file system. Technical Report UPC-DAC-RR-CAP-2010-12, Department of Computer Architecture, Universitat Politecnica de Catalunya, 2010.

[30] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposiumon Distributed Computing*, pages 284–298, Stockholm, Sweden, 2006.

[31] RMS-TM - BSC Microsoft Benchmark Suite for TM systems. Available at http://www.bscmsrc.eu/software/rms-tm.

[32] B. Saha, A.-R. Adl-tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York City, NY, 2006.

[33] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 59–66, Washington, DC, USA, 2008.

[34] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 145–155, New York, NY, USA, 2009.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, S. Margherita Ligure, Italy, 1995.

[36] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1996.

[37] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. Wormbench: A configurable workload for evaluating transactional memory systems. In *Proceedings of the 9th Workshop on Memory Performance*, pages 61–68, Toronto, ON, Canada, 2008.

[38] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: Using transactional memory in an interactive multiplayer game server. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–34, Raleigh, NC, USA, 2009.