# RMS-TM: A TRANSACTIONAL MEMORY BENCHMARK FOR RECOGNITION, MINING AND SYNTHESIS APPLICATIONS

Gökçen Kestor
Barcelona Supercomputing Center
gokcen.kestor@bsc.es

Srdjan Stipic
Barcelona Supercomputing Center
srdjan.stipic@bsc.es

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

Adrian Cristal
Barcelona Supercomputing Center
adrian.cristal@bsc.es

Mateo Valero
Barcelona Supercomputing Center
mateo.valero@bsc.es

## Abstract

Transactional Memory (TM) is a new concurrency control mechanism that aims to make parallel programming for Chip MultiProcessors (CMPs) easier. Recently, this topic has received substantial research attention with various software and hardware TM proposals and designs that promise to make TM both more efficient. These proposals are usually analyzed using existing TM-benchmarks, however the performance evaluation of TM proposals would be more solid if it included more representative benchmarks, especially from the emerging future CMP applications in the Recognition, Mining and Synthesis (RMS) domain.

In this work, we introduce RMS-TM, a new TM benchmark suite that includes selected RMS applications. Besides being non-trivial and scalable, RMS-TM applications have several important properties that make them promising candidates as good TM workloads, such as I/O operations inside critical sections, nested locking, and various percentages of time spent in atomic sections and high commit/abort rates depending on the application.

We propose a methodical process to construct a TM benchmark suite from candidate applications: in this endeavor, we divide the application selection process into static and dynamic *pre-transactification* phases and propose criteria for selecting the most suitable applications. Analyzing all the *BioBench* and *MineBench* RMS applications and applying our methodology, we selected 4 applications which form the RMS-TM benchmark suite. Our experiments show that the transactified versions of RMS-TM applications scale as well as their lock-based versions.

***Keywords*** Transactional Memory, Workload Characterization, BioBench, MineBench.

## 1. INTRODUCTION

Since it is expensive (in terms of area and power consumption) to extract more Instruction Level Parallelism (ILP) from modern processors, multicore chips have become a defacto standard, as they provide performance scalability by exploiting Thread Level Parallelism (TLP). However, the complexity of parallel programming and the difficulties of implementing efficient and provably correct programs limit the effective use of these Chip MultiProcessors (CMPs). New programming models have been proposed to ease the writing of parallel applications that perform well on multicore architectures. Transactional Memory (TM) [14] is one such programming model for control.

One could compare TM with locking, the classical concurrency control mechanism. Lock-based implementations provide consistency and isolation to threads that access shared resources, although programmers have to explicitly protect those shared resources. With TM, programmers simply mark code regions that access shared resources while the TM system provides consistency and isolation. Furthermore, TM enables programmers to write simple parallel code with coarse-grained transactions that could perform as well as parallel code that uses fine-grained locks.

Recognition, Mining, Synthesis (RMS) applications have clear relevance to mainstream workloads and have often been proposed as a good workload for future many-core processors. This has prompted us to study whether RMS applications (*BioBench* [1] and the *MineBench* [20]) benefit from using TM or not. As we show in Sections 2 and 4, the characteristics of these applications are different from the existing TM benchmarks, thus, they provide a further set of challenging test applications useful to TM researchers. For example, the applications we study involve I/O operations within critical sections, deep nesting levels, and various mix of long and short transactions.

Before selecting applications for transactification from the RMS benchmarks, we realized the we needed a methodical, well-defined procedure. Consequently, we developed a set of criteria that makes a lock-based threaded parallel program a good candidate for transactification. In a *pre-transactification* phase, we apply these criteria, such as a nested locking, complex function call traces, and irrevocable [29] [1] operations inside lock blocks, and filter out those *BioBench* and *MineBench* applications that do not generate interesting cases from a TM point of view. The selected applications are then transactified from their original lock-based parallel versions using a prototype version of Intel C++ compiler with Software Transactional Memory (STM) support [28, 7] in the *transactification phase*. Finally, we provide information about the lock-based and the TM-based implementations of the selected applications in order to provide a direct comparison to measure the benefits of TM. Moreover, we discuss the challenges faced while transactifying the applications. We validate the characteristics of the transactified applications by showing experimental results performed on a multi-core machine. According to our experimental results, the selected applications present a wide range of different transactional and runtime characteristics that qualify them as a new and comprehensive benchmark suite for evaluating TM designs. Among those properties, the most desirable and important ones are the following:

- Nested transactions (up to depth level 9) - the depth of nested transactions are unknown at compile time due to the conditional recursive function calls inside atomic blocks.
- Large amount of I/O operations, memory management operations and library calls in atomic blocks.
- Complex function calls and control flow inside atomic blocks.
- Various mix of long/short transactions with different sizes of read and write sets.
- High and low contention.

The rest of this paper is organized as follows: Section 2 summarizes other work proposed in this area and we also describe our motivation. In section 3, we introduce RMS-TM benchmark suite. The analysis and selection of the RMS-TM applications are covered in Section 4. In section 5, we show our experimental results for TM-based applications and compare their behavior against the equivalent lock-based versions. Section 6 concludes this paper and section 7 comments on future work.

[1] When a transaction runs in irrevocable mode, it is guaranteed to commit and all other transactions in the system are aborted

## 2. RELATED WORK

This section is intended to explain other benchmarks developed in recent years for analyzing parallel systems as well as TM systems.

**TM micro-benchmarks** [9] use single data structures, such as hash tables, linked lists and B-trees, to test TM implementations. These micro-benchmarks are useful for constructing basic-level insights of TM designs but do not exhibit different TM characteristics. More importantly, these benchmarks are not representative of realistic workloads, thus, they do not provide a comprehensive analysis of TM systems. In fact, realistic workloads perform operations on several, more complex data structures at the same time.

**SPLASH-2** [30] is a suite of parallel applications that consists of eight complete applications and four computational kernels. The applications and the kernels have been implemented to minimize the time spent inside critical sections. SPLASH-2 focuses on parallel applications that utilize little synchronization between threads. SPLASH-2 does not provide various sizes of critical sections or different conflict rates because of the high degree of parallelism, hence, the benchmark suite is not fully capable of evaluating the underlying TM systems and discovering interesting transactional behaviors.

**STMBench7** [12] presents an application adapted from the 007 benchmark [4] to analyze Software Transactional Memory (STM). STMBench7 provides a coarse-grained and medium-grained locking implementation in both Java and C++ that can be compared to their transactified equivalent versions. The benchmark performs complex and dynamic operations on a non-trivial data structure. However, the benchmark relies on the users to mark the critical sections with annotations that may be error-prone and time-consuming. STMBench7 performs operations only on large data structures, thus, it only shows long transactions. This characteristic is useful for evaluating virtualized transactions on TM systems. Virtualized transactions [6] are not limited in terms of execution time, memory footprint and nesting depth. Virtualization is a challenge for Hardware Transactional Memory (HTM) systems that use small hardware caches and physical addresses for transaction bookkeeping.

**STAMP** [18] is a benchmark suite that consists of several benchmarks that have various transactional and runtime behaviors. STAMP provides a sequential and a transactional version of applications but does not provide the lock-based versions.

**Lee-TM** [2] is a benchmark suite based on the Lee's routing algorithm and promises longer and realistic workloads. The benchmarks consist of sequential, coarse-grained and medium-grained lock-based, transactional and optimized transactional (with early release) implementations of Lee's routing algorithm. Thus, Lee-TM is good for comparing different locked and transactional implementations. Besides that, the data structure used in the implementation of Lee's

routing algorithm is static and all transactions are performed in two phases with very regular operations. Lee-TM has very long transactions but the benchmark is still smaller than STAMP or STMBench7.

**The Haskell STM Benchmark suite** [24] consists of 10 applications which have different code sizes (small, medium, large). All the applications in the Haskell STM suite are implemented in the Haskell functional programming language which allows us to discover different aspects of TM systems. However, the main problem is that Haskell is not suitable for HTM and hybrid TM simulation environments in terms of programming language used and almost all of these applications are also microbenchmarks.

**WormBench** [33] is a synthetic transactional application implemented by Zyulkyarov et al. in C. The inherent features of this workload are: it is highly configurable, atomic blocks are marked by supported language level keywords, and has coarse grain lock based implementation that serves as a baseline. By preparing a specific configuration, one can model the transactional behavior of an already existing parallel application or create a runtime scenario that stresses a particular aspect of the TM system. However, this synthetic application is useful mostly to mimic existing TM applications to study or debug their performance issues rather than discovering these issues and finding the patterns how transactions will be actually used within the emerging transactional applications.

While the current TM benchmarks are certainly useful, we believe there are further opportunities for developing new TM benchmarks that would serve as an application testbed for researchers that are developing solutions for current TM hot topics such as I/O and nested transactions. RMS-TM was developed with this goal in mind. Moreover, workloads with a wide range of transactional behaviors are required to avoid skewed evaluations. The TM applications in RMS-TM benchmark suite have been implemented using different algorithms and include transactions that consist of many operations on several data structures and different data access patterns. Additionally, they present varying percentage of time spent inside transactions and various abort/commit rates. TM applications with a high abort rate are good candidates for evaluating TM systems in terms of live-lock problems. Besides that, TM applications with a high commit/abort rate are useful to analyze the performance of lazy and eager data versioning used in TM systems. The RMS-TM applications have varying sizes of read and write sets which could help TM designers to decide on the correct size of the Hardware Transactional Memory (HTM) buffers. Furthermore, RMS-TM provides different input data sets, which allows us to explore different transactional behaviors by changing the size of the data sets.

The closest existing TM benchmark to RMS-TM is STAMP, which is orthogonal to RMS-TM. STAMP also has substantial applications with varying abort/commit rates and small and large transactions. In addition, RMS-TM also has I/O operations inside transactions and nested transactions, with some applications spending relatively smaller percentage of transactional runtime for small number of cores (but with the percentage increasing monotonically for higher core counts - good for many-core TM research), and with some applications in the with high transactional write/read ratios while other applications have low write/read ratios.

# 3. BENCHMARK SUITE OVERVIEW

In this section, we give a high-level overall overview of the *BioBench* and *MineBench* RMS benchmarks. We also provide an in-depth description of the applications (*Hmmer*, *Apriori*, *ScalParC*, *Utility-Mine*) that were eventually selected to be part of the RMS-TM benchmark suite. The selection process of the RMS-TM applications is covered in Section 4.

## 3.1 BioBench Benchmark Suite

*BioBench* [1] is a benchmark suite that consists of bioinformatics applications. The applications in the *BioBench* benchmarks suite use the Pthread parallel programming model [3]. These applications perform operations on textual representations of biological sequences. This behavior is opposite to the behavior of scientific applications that are expected to be floating-point intensive.

**HMMER** [8] is the sequence profile searching package selected from the *BioBench* benchmark suite. *Hmmer* aligns multiple sequences by using profile Hidden Markov Models (profile HMMs). Profile HMMs consist of statistical models of multiple sequence alignments that generally perform sensitive database searching. As opposed to homology searching, which uses either local - Smith/Waterman algorithm [26] - or global - Needleman/Wunsch algorithm [21] – alignments, *Hmmer* itself decides on which alignments to perform while building the model.

The *Hmmer* package consists of several applications, each of which has a special task to facilitate sequence profile searching. Searching an HMM database is the main functionality of *Hmmer* and is provided with two different applications: *Hmmpfam* reads each sequence from a sequence file and then compares each sequence to all the HMMs to find significantly similar sequence matches whereas *Hmmsearch* reads an HMM and searches a sequence database for significantly similar sequence matches.

In the transactional version, The *Hmmsearch's* threads read the next sequence from an input list of sequence in parallel and use transactions to protect the accesses to the input list of sequences. Moreover, the *Hmmsearch's* threads share two score lists ranked by `per-sequence` scores and `per-domain` scores and a histogram of the whole sequnce stores. Transactions are used to protect update operations on these two score lists and the histogram.

| Application | Category | Description |
|---|---|---|
| **Apriori** | ARM | Horizontal database, level-wise mining based on Apriori property |
| **Eclat** | ARM | Vertical database, Lattice transversal techniques used |
| **Utility-Mine** | ARM | Utility-based Association Rule Mining |
| **GeneNet** | Classification | Gene relationship extraction using micro-array-based method |
| **Naive Bayesian** | Classification | Simple statistical classifier |
| **Rsearch** | Classification | RNA sequence search using Context-Free Grammars |
| **ScalParC** | Classification | Decision Tree Classification |
| **SEMPHY** | Classification | Gene sequencing using phylogenetic tree-based method |
| **SNP** | Classification | Hill-climbing search method for DNA dependency extraction |
| **SVM-RFE** | Classification | Gene expression classifier using recursive feature elimination |
| **BIRCH** | Clustering | Hierarchical Clustering method |
| **K-Means** | Clustering | Mean-based data partitioning method |
| **Fuzzy K-Means** | Clustering | Fuzzy logic-based data partitioning method |
| **HOP** | Clustering | Density-based grouping method |
| **PLSA** | Optimization | DNA sequence alignment using Smith-Waterman optimization method |

**Table 1.** Overview of the MineBench Benchmark Suite [20]

### 3.2 MineBench Benchmark Suite

Data mining is a technique to extract meaningful information from vast raw data. Recently, applications that use a data mining technique have become important in both commercial and scientific areas. Data mining applications can be categorized into the following: classification, clustering, Association Rule Mining (ARM), data virtualization, sequence mining, similarity search and optimization.

The *MineBench* benchmark suite [20] has been designed considering the data mining categories that are commonly used in today's industry problems and are likely to be used in the future. This property strongly motivates us to work on this benchmark suite. As shown in Table 1, the applications from the classification, clustering, Association Rule Mining (ARM) and optimization (similarity search) categories represent the heterogeneity of the algorithms and are implemented using OpenMP [5, 22] as the parallel programming model.

A classification algorithm builds a model to assign unclassified records in the test dataset to one of several predefined categories or classes. A clustering algorithm partitions the data set into subsets containing elements of similar properties. Association Rule Mining (ARM) identifies interesting relationships uncovered among the large number of transactional records. Sequence alignment is one of the most commonly used technique to perform similarity searching in bioinformatics. It is typically used to compare one query nucleotide (DNA or RNA) or a protein sequence against a database of sequences, and to discover similarities and sequence matches.

**Apriori** [31] is a well-known Association Rule Mining algorithm performed on transactional records in a database. *Apriori* uses downward closure properties (top-down search): at every iteration, during the candidate generation step, only the itemsets that meet a minimum support criterion in the previous iteration generate a new candidate set. *Apriori* checks whether all the subsets of the itemset have a required support value before inserting an itemset into a new candidate set. This step is referred to as the pruning step and is used to eliminate any candidate whose subset include at least one element with low minimum support. *Apriori* uses a hash tree to store candidates. An internal node at depth $d$ in the hash tree has a hash table which shows the nodes at depth $d+1$ while the leaves hold the itemsets. The algorithm terminates when no itemsets can be added to a candidate set.

Transactions are used to protect the calculation of support values and the insertion of a candidate itemset into the hash tree. Since the counts of the itemsets in the hash tree are common, more than one thread may try to access the counter value and increment it. For example, a thread travels the hash tree starting from the root node and hashes on successive items in the itemsets until it reaches a leaf node. At this point, thread `i` may meet another thread which also wants to increment the itemset.

**ScalParC** [15] 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 value of attributes. This process continues until each record entirely consists of examples from one class. *ScalParC* uses a distributed hash table in the splitting phase of the decision tree algorithm; a new parallel hashing paradigm has been defined to build and access this hash table. Due to the parallel hashing paradigm, *ScalParC* demonstrates a scalable behavior in both runtime and memory requirements. During the attributes partitioning phase, different threads try to simultaneously access a shared counter, which is used to schedule attributes dynamically. Transactions protect the access to this shared counter.

**Utility-Mine** [16] is another ARM technique. A utility mining model has been developed to identify itemsets with high utilities. The utility of an item or an itemset can be defined as how useful the item or the itemset is. The utility mining algorithm consists of two phases: In the first phase, a "transaction-weighted downward closure property" is used to generate candidate itemsets on the search space. In the

| Application | Domain | Synchronization | Locking type | Nested Lock. | Func. call | Special operations in CS[a] |
|---|---|---|---|---|---|---|
| **Hmmersearch** | Sequence profile searching | No | Coarse-Grained | No | Yes | Large amount of I/O, memory management operations, lib. calls |
| **Hmmerpfam** | Sequence profile searching | No | Coarse-Grained | No | Yes | Large amount of I/O, memory management operations, lib. calls |
| **Apriori** | Association Rule Mining | Yes | Coarse-Grained Fine-Grained | Yes | Yes | Memory management operations |
| **GeneNet** | Bayesian Network | No | Fine-Grained | No | No | Memory management operations |
| **Kmeans** | Clustering | No | Fine-Grained | No | No | No |
| **Fuzzy-Kmeans** | Clustering | No | Fine-Grained | No | No | No |
| **PLSA** | Dynamic Programing | No | Fine-Grained | No | Yes | No |
| **Rsearch** | Patern Recognition Mining | No | Fine-Grained | No | Yes | Memory management operations |
| **ScalParC** | Classification | Yes | Coarse-Grained Fine-Grained | No | No | No |
| **Utility-Mine** | Association Rule Mining | Yes | Coarse-Grained Fine-Grained | Yes | Yes | Memory management operations |

[a] CS: Critical Section

**Table 2.** Benchmarks in Static Pre-Transactification phase

second phase, the high utility itemsets are defined by scanning the transaction database.

A single common hash tree, shared by all the processors, stores the candidate itemsets at each level of search as well as their transaction-weight utilization. In the TM version included in RMS-TM, transactions protect the update of the utility of itemset and insertion of a candidate into the tree.

# 4. THE TRANSACTIFYING PROCESS

The main goal of our study is to define a set of representative TM benchmarks to cover many different transactional execution scenarios. In order to characterize the applications comprehensively, we first determined the set of criteria that we evaluated during each transactification phase. Then, we analyzed the applications included in the suites, *BioBench* and *MineBench*, to test whether they are suitable for evaluating TM systems with respect to our evaluation criteria. This process, described in Section 4.1, is referred to as the *pre-transactification* phase. Before the *pre-transactification* phase, we examined the applications in terms of their programming methodology. Candidate applications for our new benchmark suite should be from the shared memory domain, and hence, implemented using an implicit/explicit lock-based parallel programming model. Consequently, embarrassingly parallel as well as serial applications were discarded. The rest of the applications qualified as candidates to be analyzed in the *pre-transactification* phase. The applications that were considered good candidates in the *pre-transactification* phase were transactified, as described in section 4.1, using the Intel C++ compiler with Software Transactional Memory support in the transactification phase.

## 4.1 Pre-Transactification Phase

It is crucial to choose the set of applications according to TM-specific "usefulness" criteria such as nested transactions, various abort/commit rate, irrevocable operations inside atomic blocks, etc. To achieve this goal, we analyzed the RMS application domain and chose only the applications that are promising from the TM point of view according to the defined evaluation criteria.

The *pre-transactification* phase consists of two sub phases: *static* and *dynamic*. Dividing the *pre-transactification* phase into two phases helped us to make an effective and a comprehensive analysis on the whole application domain. Several criteria have been evaluated in each transactification phase/sub-phase. These evaluation criteria allow us to reduce the number of applications to be examined in the benchmark suites and, at the same time, to maintain enough applications to capture different applications' characteristics and to ensure that the evaluation of future TM design is exhaustive.

### 4.1.1 Static Pre-Transactification Phase

The evaluation criteria used in the static *pre-transactification* phase are listed below:

- Synchronization constructs used between lock blocks such as barriers, wait or join. Alternative synchronization constructs like barrier could likely cause high contention since all the transactions attempt to enter the atomic block (e.g., the barrier) at the same time but only one will commit.
- The type of locking granularity (fine-grained locking or coarse-grained locking). Fine/coarse grained critical sections inside a lock-based application lead to small/large transactions respectively, i.e., the locking granularity is related to the transactions' length.
- Nested locking. Managing read- and write-set is more complicated with nested transactions because whenever an inner transaction ends (either commits or aborts), its

| Application | Number of Threads | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| **hmmsearch** | 0.30 | 0.36 | 0.37 | 0.50 |
| **hmmpfam** | 37.36 | 36.83 | 37.86 | 37.21 |
| **Apriori-s** | 0.01 | 0.01 | 0.02 | 0.02 |
| **Apriori-m** | 0.03 | 0.05 | 0.07 | 0.37 |
| **Apriori-l** | 0.02 | 0.02 | 0.04 | 0.10 |
| **PLSA** | 0.0026 | 0.0071 | 0.0077 | 0.028 |
| **Rsearch** | 0.0001 | 0.0003 | 0.0004 | 0.0004 |
| **ScalParC-s** | 0.05 | 0.24 | 0.39 | 0.14 |
| **ScalParC-m** | 0.04 | 0.18 | 0.26 | 0.11 |
| **ScalParC-l** | 0.02 | 0.09 | 0.19 | 0.27 |
| **UtilityMine-r** | 10.55 | 10.26 | 7.94 | 6.42 |
| **UtilityMine-s** | 64.23 | 59.04 | 54.69 | 52.22 |
| **UtilityMine-m** | 82.99 | 76.81 | 69.27 | 67.72 |

**Table 3.** Time in Critical Sect. for Lock-based Apps (in %)

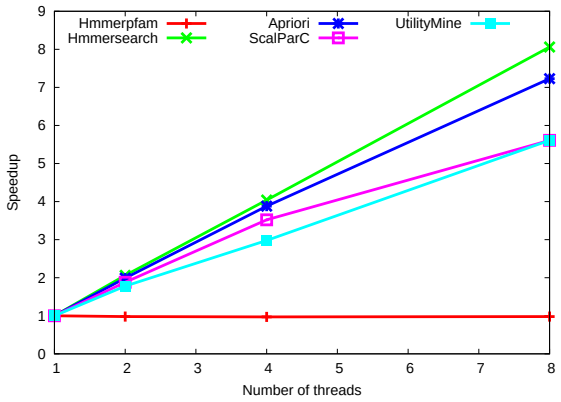


**Figure 1.** Scalability of Lock-based Apps (Large Dataset)

changes may eventually be visible to the outter transactions.

- Function calls between acquiring and releasing locks. Functions calls inside transactions increase the complexity of the transactions because it is not easy to determine the length (in terms of execution time) of the transaction at compile time. Some control flow instructions, like loops, may create the same situation.
- Special operations inside critical sections, such as I/O operations, library calls or dynamic memory management operations (malloc, calloc, free). Some STM compilers may support some or all of those special operations. For example, the Intel STM compiler supports dynamic memory management operations but does not support I/O operations. On the other hand, at this stage, the GCC-TM [10] compiler does not support dynamic memory allocation. All those special operations not supported by the STM compilers are likely to be marked as irrevocable and create an impact on the STM system performance.

Again, the goal of this phase is to select representative applications that cover all the exploration space with their properties. So, we did not only select applications that spend a large amount of time in critical sections, but also applications that spend a relatively short time in a critical section. For example, applications that spend only a small amount of execution time inside critical sections (thus, potentially inappropriate for TM analysis) are included in the benchmark suite if they have other irrelevant properties, such as I/O operations inside the transactions. Notice that our usefulness criteria are not specific to the RMS applications but are application-agnostic and can be used as a general tool to determine the applicability of any application to the TM design.

We discarded embarrassingly parallel and serial applications even before the *pre-transactification* phase. *Eclat, Bayesian Network, Birch, Semphy* are discarded because they are serial applications. *HOP* and *SNP* are discarded because they are embarrassingly parallel with each thread performing independent operations of the others. In Zambreno et al. [32] the authors show that *SVM-RFE* scales poorly due to redundant locking of memory structures. *SVM-RFE* could have a better scalability if the lock could be removed, hence, *SVM-RFE* could be a good candidate for TM system evaluation. However, the application does not use the locks explicitly in the source code: the locks are used in the Intel Math Kernel Library while calling the clock function from this library for time measurements. Therefore, *SVM-RFE* was not selected to be analyzed in the *pre-transactification* phase. The applications in the *MineBench* suite are more diverse than the ones in the *BioBench* suite: although different in size, *BioBench* applications are mainly from the domain of gene sequence similarity. Therefore, we selected *Hmmer* which is the more popular and representative application from this domain.

Table 2 shows the evaluation results of the *static pre-transactification* phase for the applications that passed the preliminary evaluation step. The selected applications were passed to the *dynamic pre-transactification* phase. *Hmmpfam* and *Hmmsearch* from the *Hmmer* package are special cases: we selected these applications because they exhibit a large amount of I/O, memory management operations and relatively complex function calls in critical sections. Moreover, they have an unstructured use of locks. This means that a lock may be released in several places, depending on the control flow: this makes transactifying these kind of benchmarks challenging. However, they show different TM characteristics because the length of transaction changes at runtime depending on the lock releasing condition. The other reason to select these applications is that the benchmarks are implemented with coarse-grained locking and they present a large number of instructions in the critical sections. In fact, applications that have a coarse-grained locking structure are good candidates for our study because the coarse-grained lock-based programming causes threads to waste a significantly high amount of time in the synchronization part of

the application; Minimizing this synchronization time is an important topic for TM research.

Although *Apriori* and *Utility-Mine* are in the same domain, they both satisfy other evaluation criteria (see Table 2). The reason to select *ScalParC* is that the benchmark comes from a different application domain. In particular, *ScalParC* includes both fine-grained and coarse-grained locking, meaning that it has different types and sizes of transactions. In addition, *ScalParC*, *Apriori* and *Utility-Mine* use synchronization constructs between lock blocks as shown in Figure 2. We expected that the placement of synchronization constructs between lock blocks could create interesting TM characteristics such as a somewhat high abort rate even when the application does not spend much of its execution time in transactions.

```
atomic{
  statement1
}
#pragma omp barrier
atomic{
  statement2
}
```

**Figure 2.** Usage of Synchronization constructs

*PLSA* and *Rsearch* are the other two applications that passed the *static pre-transactification* evaluation step. Besides being from a different application domain, they have function call(s) inside critical sections. Calling function(s) within atomic blocks makes it difficult to statically determine how long transactions and their read/write sets are. Moreover, Rsearch performs memory management operations in critical sections. As we mentioned in section 4.1.1, not all the avaliable STM compilers support dynamic memory allocation inside critical sections. In these cases, those operations are marked as irrevocable operations.

*Kmeans* and *Fuzzy Kmeans* are clustering applications in the same package. They are very small applications that do not present nested locking, function call or irrevocable operations inside critical sections, as well as synchronization constructs between lock blocks. We discarded these applications from our benchmark suite.

Although *GeneNet* is a realistic and comprehensive application, it has small critical sections with few memory operations and no function calls. Therefore, *GeneNet* was not selected as a candidate for *dynamic pre-transactification* phase.

#### 4.1.2 Dynamic Pre-Transactification Phase

In the *dynamic pre-transactification* phase, we used scalability (shown in Figure 1) and time spent within critical sections (shown in Table 3) as evaluation creteria. To measure the time spent inside critical sections, we instrumented the applications that had passed the *static pre-transactification* phase successfully. Most applications in the original benchmark suites, have been carefully implemented by expert programmers to achieve the highest parallelism. While this is, in general, a desirable property, the resulting small critical sections translate into TM programs with small transactions. Due to the short transaction length and the low transaction frequency or the small size of read and write set, those applications are unable to stress the underlying TM systems. *PLSA* and *Rsearch* follow this class of applications which means they spend a slightly short time within critical sections. These results confirm our claim in the *static pre-transactification* phase, thus, we discarded them in the *dynamic pre-transactification* phase. Even though ScalParc and *Apriori* spend a short time inside critical sections, they have several marked atomic blocks and use synchronization constructs, such as barriers between consecutive atomic blocks. We believe that these characteristics could result in quite interesting TM behaviors such as high abort rate.

*Apriori* and *Utility-Mine* have a high level of nested locking built by recursive function calls as shown in Figure 3. As mentioned before, an insertion of a candidate itemset is enclosed in a transaction. At the insertion point, if the number of itemsets at a leaf node exceeds a threshold value, that leaf is converted into an internal node by the rehash function. Their nested depth reaches up to 9, depending on the data set size. As a result, *Utility-Mine* and *Apriori* are good TM benchmarks to evaluate the TM systems with support for arbitrary levels of nested transactions.

```
rehash()
{
  add_element_to_hash();
}
add_element_to_hash()
{
     atomic{
      if (numitems > threshold)
       rehash();
     }
}
```

**Figure 3.** Nested locking

In the *Hmmer* sequence profile searching package, we selected *hmmsearch* even if it spends a short time inside critical sections. However, this application scales better than *hmmpfam*, as shown in Figure 1. The figure shows that only *hmmsearch* presents a linear speedup, the other benchmarks have a sub-linear speedup. *Hmmpfam* is dominated by I/O operations and essentially behaves like a serial application; *Apriori*, *ScalParC* and *Utility-Mine* scale well except when they run on 8 threads in parallel (all the available processors in our experimental setup) especially with small or medium data sets. In this case the speedup is smaller (or much smaller, as for *ScalParC*) than the speedup with 4 threads. The reason for the performance loss on some of the applications is not related to accessing shared data structures. In [17] the authors conclude that load balancing and Operating System (OS) noise are the main culprits for the limited speedup of certain. Load balancing (or data load bal-

| Application | Thread Number | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| **hmmsearch** | 0.44 | 0.51 | 0.54 | 0.67 |
| **TM-Apriori-s** | 0.01 | 0.02 | 0.03 | 0.36 |
| **TM-Apriori-m** | 0.13 | 0.18 | 0.25 | 0.43 |
| **TM-Apriori-l** | 0.06 | 0.09 | 0.13 | 0.31 |
| **TM-ScalParC-s** | 0.10 | 0.38 | 1.22 | 2.75 |
| **TM-ScalParC-m** | 0.07 | 0.28 | 0.79 | 2.31 |
| **TM-ScalParC-l** | 0.04 | 0.13 | 0.57 | 3.90 |
| **TM-UtilityMine-r** | 66.89 | 68.42 | 65.6 | 61.52 |
| **TM-UtilityMine-s** | 97.36 | 97.19 | 97.19 | 97.14 |
| **TM-UtilityMine-m** | 97.96 | 98.75 | 98.85 | 98.88 |

**Table 4.** Time in Atomic Block for TM-based apps (in %)

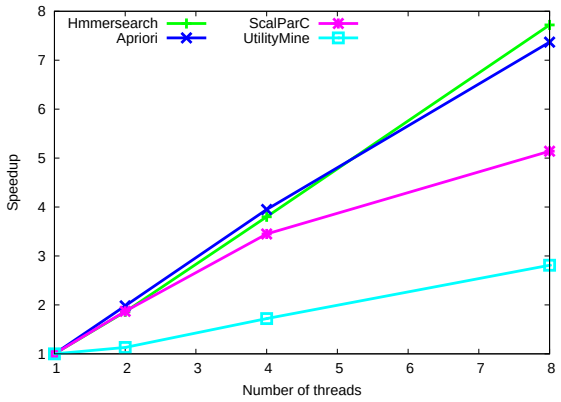


**Figure 4.** Scalability of TM-based Applications (Large Dataset)

ancing) is a common problem with Single Program/Multiple Data (SPMD) parallel applications.

In SPMD applications, each thread executes the same operation on its portion of the data set and then reaches the synchronization point. If the amount of data that each thread's process loads is not balanced, threads with a larger amount of data to process will reach the synchronization point after the other less-loaded threads, actually limiting the performance of the parallel applications, no matter how many processors are used. OS noise [25, 27, 11] appears when the OS needs to perform some operations and temporarily removes one of more threads of the parallel application from their CPUs. At the end of the system activity, the thread is rescheduled for execution by the OS but the delay introduced by the OS will reduce the performance of the whole application. In fact, no matter how well balanced the load is, if one thread suffers from OS noise, it will reach the synchronization point after the others, becoming the bottleneck of the entire application. If the parallel application is not using all the available processors in the system, the OS tries to schedule system activities on the free CPUs. On the other hand, if the application is using all the available processors (8 in our case), the OS has no choice but to stop one or more threads and resume their execution after the system activities have completed. As a result, both data load balancing and OS noise may cause performance degradation especially when using a large number of threads.

### 4.2 Transactification Phase and Challenges

In this section, we describe how to transactify lock-based applications considering the programming effort. We also identify the challenges faced during the transactification phase. Transactifying applications from their equivalent lock-based versions was not a straightforward task because each application has a different structure. We first analyzed each application to understand the parallelization strategy and the use of locks for synchronizations. In order to maintain the original semantics of applications we kept the size of the atomic blocks as the ones in the lock-based versions during the transactification phase.

To transactify the applications selected as good candidates in the *pre-transactification* phase, we replaced locks used to protect the accesses to shared variables with transactions using a prototype of the Intel C/C++ compiler with STM support. Intel STM compiler provides simple language extension to develop TM applications. However, to apply the transactional semantics to the functions inside atomic blocks, the functions should be marked as either `TM-callable`[2] or `TM-pure`[3]. If other kind of functions are called inside the atomic block, the compiler might make the transactions run irrevocably.

The applications we examined often allocate objects and/or call external functions inside atomic blocks. The objects are allocated through the `new` operator which, in this version of the Intel STM compiler, appears not to be marked as `TM-callable`. The `new` operator, in turn, calls the object constructor but, at this point, the transaction will run irrevocably even if the object constuctor is marked as `TM-callable`. This happens because the first function called inside the transaction (the `new` operator) is not marked as `TM-callable`.

In order to deal with this challenge and avoid the transactions to run irrevocably, we overloaded the `new` operator and marked it as `TM-callable`. The Intel STM compiler generates a TM-clone version of the `TM-callable` function and allows the user to safely call this function inside an atomic section. In this specific case, whenever the `new` opearator is called inside an atomic block, the application calls the overloaded operator (marked as `TM-callable`), instead of the standard one (not marked as `TM-callable`).

Other challenges concern acquiring and releasing locks in an unstructured way, meaning that locks can be released in more than one point in the source code. Transactifying an

[2] Compiler generates a clone of function annotated as TM `callable` and translate each memory `read` and `write` to a `TM read barrier function` and a `TM write barrier function`

[3] A programmer guarantes that a function marked as `TM-pure` does not access shared variables when it calls inside transactions.

application with an unstructured lock structure is not easy because there is syntax difference between locking programming and TM programming with the Intel STM compiler. To deal with this challenge, we used the same technique used by [34, 18].

## 5. EXPERIMENTAL RESULTS

In this section we compare the applications in RMS-TM to their equivalent lock-based versions and show how the selected applications cover all the characteristics of the TM systems. We report our experimental results for scalability, time spent in atomic sections and transaction abort and commit rates. In the experiments, we report the average results of five running executions using three different data set sizes: Small (s), Medium (m) and Large (l) for *Apriori* and *ScalParC*; Real (r), Synthetic Small (s) and Synthetic Medium (m) for UtilityMine.

We performed our experiments on Dell PE6850 workstation with 4 dual core x64 Intel Xeon processors runing at 3.2Ghz equiped with 32KB IL1 and 32KB DL1 private per core, 4MB L2 shared between the two cores on die, 8MB L3 shared between all cores, and 32GB RAM. The installed operating system was SUSE 11.0.

Figure 4 shows the scalability of the applications. All the TM applications (except *Utility-Mine*) present a scalability similar to their equivalent lock-based versions. Several factors may influence the scalability of *Utility-Mine*: for example, the number of aborted transactions, which is, arguably, the most common reason for poor scalability. As seen in Table 5, although *Utility-Mine* has a low abort rate, the benchmark presents a large number of transactions, each one with large read/write sets. This means that every rollback operation is expensive (i.e. the cost of each rollback depends on the amount of data, not on the number of rollbacks), thus, affecting performance. To understand another impact on the performance loss, we performed a deeper analysis of all the applications using the `oprofile` [23] system profiler and looked at specific performance counters. Unlike all the other applications, *Utility-Mine* has large read- and write-sets; this causes more accesses to memory and, therefore, more L2 cache misses. In fact, we notice that, when running this benchmark, more then 90% of the L2 cache misses are caused by the Intel STM library when accessing the read- and write-sets of the application. This extra overhead is larger as the sizes of the read- and write- sets increase, thus, it limits the scalability of the application. However, having large read- and write-sets is a good property for a TM benchmark since this property enables researchers to evaluate TM systems with support for virtualized transactions. This is in agreement with what is stated in [34]. In general, however, we experience that TM does not significantly influence the scalability.

Table 4 presents the time spent in atomic blocks with respect to the total execution time with 1, 2, 4 and 8 threads for each data set. We expected to have some overhead introduced by the Intel STM compiler and run-time library because of the extra work required by the STM system to handle transactions, for example, when detecting conflicts. Moreover, as we can see comparing Table 3 to Table 4, the Intel STM mechanism introduces different overheads in the transactified versions of the benchmarks. For example, the lock version of *TM-Apriori-m* spends 0.37% of the time inside critical sections and 0.47% in the transactified version. On the other hand, *Utility-Mine-r* spends 6.42% of its time in critical section with the lock implementation and 61.52% with the TM implementation. Rollback on abort is a major source of performance overhead in the *TM-Utility-Mine* benchmark because the benchmark, with 8 threads, spends 66% of its total atomic time for wasted work [24]. As we've already explained, this increment is not only caused by rollbacks but also by the larger number of L2 cache misses introduced by the read- and write-sets. Looking at Table 4, we can see that the selected 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 both applications that are very sensitive to TM overhead (like *TM-UtilityMine-r*), and an application that is not sensitive to the TM systems' overhead (*TM-Apriori-m*). Notice that *TM-UtilityMine* spends most of its execution time inside critical sections, which is important for analyzing TM systems with support for virtualized transactions.

Table 5 summarizes the transactional characteristics of the RMS-TM applications. As we can see from the table, the RMS-TM benchmark suite explores several combinations: small read/write sets (short transactions) with a high abort rate (*TM-Apriori*); large read/write sets (long transactions) with a limited abort rate (*TM-Utility-Mine*); small read/write set with medium abort rate (*TM-ScalParC*), etc. Our experiments confirm what we observed in the *static pre-transactification* phase: for example, *TM-Apriori* and *TM-ScalParC* access shared counters inside the atomic blocks and use points of synchronization, which results in threads to enter a transaction at the same time, leading to increased contention among threads. Therefore, those two applications have high contention, even if they spend most of their execution time outside atomic blocks. From this analysis, we can conclude that *TM-Utility-Mine* (high commit rate), *TM-Apriori* and *TM-ScalParC* (high abort rate) are good candidates for the evaluation of both lazy and eager data versioning. According to [13, 19], high commit/abort rates have a large impact on the performance of eager/lazy data versioning in hardware TM systems. This happens because eager data versioning relies on the idea that the commit rate is higher than the abort rate in TM benchmarks. Therefore, these systems are designed with a low commit cost. On the contrary, in hardware TM systems with lazy data versioning, the abort cost is significantly lower than the commit cost.

| Application | Read Set (in bytes) | | | | Write Set (in bytes) | | | | Transactions | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Min | Avg | Max | Total | Min | Avg | Max | Total | #Commit | #Abort | Abort % |
| **hmmersearch** | 12 | 30 | 76 | 9,208K | 0 | 2 | 4 | 613K | 306,658 | 0 | 0 |
| **TM-Apriori-s** | 20 | 153 | 1,448 | 10K | 2 | 89 | 697 | 6K | 70 | 85 | 54 |
| **TM-Apriori-m** | 20 | 461 | 19,128 | 6,580K | 0 | 296 | 10,356 | 4,228K | 14,264 | 32,990 | 69 |
| **TM-Apriori-l** | 20 | 455 | 18,163 | 6,577K | 0 | 291 | 7,920 | 4,213K | 14,439 | 33,520 | 69 |
| **TM-ScalParc-s** | 28 | 99 | 2,924 | 4,976K | 1 | 6 | 204 | 322K | 50,236 | 44,882 | 47 |
| **TM-ScalParc-m** | 28 | 103 | 3,647 | 7,725K | 1 | 6 | 252 | 503K | 74,544 | 72,680 | 49 |
| **TM-ScalParc-l** | 28 | 115 | 3,650 | 8,689K | 1 | 7 | 249 | 568K | 75,408 | 89,280 | 54 |
| **TM-UtilityMine-r** | 20 | 2,409 | 235,674 | 10,913,794K | 4 | 6 | 364 | 30,390K | 4,529,132 | 261,502 | 5 |
| **TM-UtilityMine-s** | 44 | 9,033 | 837,661 | 394,986,059K | 4 | 7 | 247 | 321,696K | 43,724,391 | 425,827 | 1 |
| **TM-UtilityMine-m** | 20 | 14,303 | 2,414,583 | 5,655,782,175K | 1 | 7 | 1716 | 3,006,263K | 395,425,752 | 3,580,362 | 1 |

**Table 5.** Transactional Behavior of TM Benchmarks with 8 threads

For software TM systems, there is no evidence about whether commit and abort rates are important metrics to analyze the performance of eager/lazy data versioning systems. In order to allow researchers to perform an exhaustive study of software TM systems, RMS-TM covers both high and low commit rates.

*TM-Hmmsearch* has irrevocable operations, such as I/O operations inside the atomic blocks; in this case, the Intel STM compiler generates irrevocable transactions and, as a result, the benchmark has no aborted transactions. We are currently working on this challenge and we plan to propose a solution as future work.

Another useful TM characteristic is the write-set/read-set size ratio. The higher the ratio, the more the probability of conflicts. RMS-TM applications have low (*TM-UtilityMine*), medium (*TM-ScalParC*) and high (*TM-Apriori*) write-set/readset ratios.

## 6. Conclusions

Transactional Memory (TM) is a new concurrency control mechanism that promises to ease CMP parallel programming while providing reasonable performance and scalability. Several TM systems (Hardware, Software and Hybrid) have been proposed in previous work. However, most of those proposals have been tested on simulators and there is no agreement on what are the important and representative TM benchmarks to run in order to thoroughly explore them.

In this paper we present RMS-TM; a new TM benchmark suite that includes scalable future CMP workloads from the Recognition, Mining and Synthesis domain. We develop a general methodology to select the best candidate applications for a TM-benchmark suite. In our methodology, we first highlight the desirable characteristics of a TM benchmark suite and define the criteria that can be used to select well representative applications for studying TM systems. Then we analyze the current TM benchmarks from the RMS domain and select the ones that satisfies our criteria (*pre-transactification* phase). Once the benchmarks had been selected, we transactified them using the Intel STM compiler. In this phase, (*transactification phase*) we faced and solved several challenges and reported the effort required by a programmer to transactify parallel applications. This contribution is important because it is always hard to quantify the programmer effort and to identify the challenges that must be solved in order to transactify an application. We hope that our experience will be helpful to other TM programmers.

Our experiments show that the transactified RMS-TM applications have very good scalability and are, thus, good candidates for extending TM research to many-core CMPs.

## 7. Future Work

As future work, we plan to port the RMS-TM applications to other STMs and to selected HTMs such as Log-TM and TCC. In time, other RMS benchmarks such as BioParallel might be analyzed and more applications might be added to the RMS-TM suite. Finally, we believe that parallelization of the existing sequential applications in the RMS domain using TM might be valuable to understand programmability aspects of TM.

## Acknowledgments

We would like to thank Intel for making the prototype C++ STM Compiler publicly available. We would like to thank the anonymous reviewers and Tim Harris, Roberto Gioiosa, Nehir Sonmez, Gülay Yalçın and Ferad Zyulkyarov for their useful comments and also Berkin Ozisikyilmaz for his help with *MineBench*. 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 (HiPEAC) and by the European Commission FP7 project VELOX (216852).

## References

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *ISPASS '05*, 2005.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Lee-tm: A non-trivial benchmark for transactional memory. In *ICA3PP '08*, June.

[3] David R. Butenhof. *Programming with POSIX threads*. 1997.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *SIGMOD '93*, 1993.

[5] B. Chapman, G. Jost, and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. 2007.

[6] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, V. Biesbrouck, M., G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. *SIGPLAN Not.*, 2006.

[7] Intel Corporation. In *Intel C++ STM Compiler Prototype Edition 2.0 Language Extensions and Users Guide*, 2008 March.

[8] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 1998.

[9] J. Ennals, R. *Adaptive Evaluation of Non-Strict Programs. PhD thesis*. July 2007.

[10] GCC. In *Transactional Memory Support for GCC*, 2008.

[11] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *ISSPIT '04*, December.

[12] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS '07*, 2007.

[13] 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 *ISCA '04*, 2004.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[15] M. V. Joshi and G. Karypis. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *IPDPS '98*, 1998.

[16] Y. Liu, W. Liao, and A. Choudhary. A fast high utility itemsets mining algorithm. In *UBDM '05*, 2005.

[17] Y. Liu, J. Pisharath, W. Liao, G. Memik, A. Choudhary, and P. Dubey. Performance evaluation and characterization of scalable data mining algorithms. In *the 16th PDCS '04*, 2004.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08*, September.

[19] E. Moore, K., J. Bobba, J. Moravan, M., D. Hill, M., and A. Wood, D. Logtm: Log-based transactional memory. In *HPCA '06*. 2006.

[20] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on*, 2006.

[21] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 1970.

[22] The openmp api specification for parallel programming. Available at http://www.openmp.org.

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

[24] 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 *CF '08*, 2008.

[25] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *ACM/IEEE SC '03*, 2006.

[26] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. 1981.

[27] D. Tsafrir, Y. Etsion, G. Feitelson, D., and S. Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *ICS '05*, 2005.

[28] C. Wang, W. Chen, Y. Wu, B. Saha, and A. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07*, 2007.

[29] A. Welc, B. Saha, and A. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08*, 2008.

[30] C. Woo, S., M. Ohara, E. Torrie, P. Singh, J., and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA '95*, 1995.

[31] J. Zaki, M., M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multiprocessors. In *KAIS*, 1996.

[32] J. Zambreno, B. Ozisikyilmaz, G. Memik, and A. Choudhary. Performance characterization of data mining applications using minebench. In *In 9th CAECW*, 2006.

[33] F. Zyulkyarov, S. Cvijic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Wormbench - a configurable workload for evaluating transactional memory systems. In *MEDEA '08*, 2008.

[34] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic quake: Use case of transactional memory in an interactive multiplayer game server. In *PPoPP '09*, 2009.