# A Study of Factors External to Software Transactional Memories that Affect their Speed-up

David Aparicio
IC, Unicamp
Campinas, Brasil
ra142289@students.ic.unicamp.br

Luiz E. Buzato
IC, Unicamp
Campinas, Brasil
buzato@ic.unicamp.br

April 6, 2014

**Abstract**

This project seeks a characterization of factors external to software transactional memories (STM) that may affect their performance (speed-up). The characterization can help the researchers community to settle down once and for all whether or not a particular selection of kernel parameters has a significative effect on the results reported by the benchmarks. In this project we use three benchmarks—STAMP, STMBench7, and eigenBench—to assess the behaviour of two STMs—TinySTM and SwissSTM—when external factors, derived from different operating system features, are switched on and off in a series of experiments. The external factors are turned on and off by changing parameters of the process manager (scheduler), memory manager, and by binding the TM implementations to different thread libraries.

## 1 Introduction

The concept of atomic transaction can be considered one of the essential concepts of Computer Science [4]. In fact, the use of transactions in database systems has been seen as a key step for the success for relational databases. They allow to use of more complex operations concurrently, while maintaining consistency of reads and writes. Thanks to transactions database management, systems have been able to consistently scale up their performance over they years.

The advent of multicore processors has created the equivalent of the concurrency harnessing problem for applications in general, because multicore computers allow the parallel execution of multiple threads that com-

municate by concurrently reading and writing to shared memory. In 1993, Herlihy and Shavit [6] proposed an adaptation of the transaction concept, namely transactional memory (TM), as a way of trying to solve the concurrency problem for multicore machines. Since then, several implementations of transactional memory appeared in hardware (HTM), software (STM), and hybrid, using a combination of mechanisms implemented in hardware and/or software. As the number of transactional memory implementations grew so did the necessity to compare their relative performances. To address this issue, researchers have created transactional memory benchmarks. Despite the existence of several benchmarks and transactional memory implementations, there is a question that is still open to further research:

For a given combination of STM and benchmark, is there a parameter that can be changed in one of the components of the environment—hardware or operating system—that has a significant impact upon the results (speed-up) of the benchmark?

To make the problem concrete let us consider two examples. The selected combination of benchmark and STM is STMBench7 and TinySTM [3], respectively. The environment is composed of an Intel machine with 16 cores and the operating system is a Linux-based distribution equipped with the latest stable version of the Linux kernel. As a first example, the parameter whose influence upon the benchmark results the experiments want to evaluate is the task affinity (process affinity with a certain core or pool of cores) of the process scheduler. So, if the scheduler's process affinity is changed, what is going to happen to the speed-ups reported by the benchmark? For the second example, we retain STMBench7 and TinySTM but run them with different configurations of thread libraries: LinuxThreads versus NTPL.

# 2 Software Transactional Memory Implementations

This section contains a summary of the software transactional memories used in this study: TinySTM [3] [`tmware.org`] and SwissTM [2] [`lpd.epfl.ch`]. Further details about both TMs can be found in the respective sites.

## 2.1 TinySTM

TinySTM is a word-based STM that uses locks to protect shared memory locations. It has been used as the basis for a number of experiments on transactional memory because of its very simple contention manager based on a encounter-time locking with the following potential advantages:

- early detection of conflicts between transactions can increase throughput because transactions do no per-

form useless work. By contrast, commit-time locking may help so avoid some read-write conflicts, but general conflicts discovered at commit time cannot be solved without aborting at least one transaction;

- encounter-time locking allows tinySTM to efficiently handle reads-after-writes without requiring expensive or complex mechanisms.

From the point of view of recovery managament, tinySTM implements two techniques: (i) write-through, and (ii) write-back. With write-through access, updates are written directly to memory and previous values are stored in an *undo* log to be reinstated in the case of an abort. With write-back access, updates are stored in a *write log*, and written to memory upon commit.

Summarily, tinySTM is a resonably simple STM implementation that has a good potential for a comparative study such as the one devised for this project.

## 2.2 SwissTM

In relation to TinySTM, the main distinctive features of SwissTM are:

- A conflict detection mechanism that detects (i) write/write conflicts eagerly, in order to prevent transactions that are doomed to abort from running and wasting resources, and (ii) read/write conflicts late, in order to optimistically allow more parallelism between transactions. Intuitively, transactions eagerly acquire objects (locks) for writing, which helps to detect write/write conflicts as soon as they appear. This also avoids wasting work of transactions that are already doomed to abort after a write/write conflict. The use of invisible reads allows transactions to read objects acquired for writing with the property that the detection of read/write conflicts is delayed, thus increasing the potential for inter-transaction parallelism. A time-based scheme is used to reduce the cost of transaction validation with invisible reads.

- Contention management uses a two-phase protocol that has no overhead on read-only and short read-write transactions while favoring the progress of transactions that have performed a significant number of updates. Basically, transactions that are short or read-only use the simple but inexpensive timid contention management scheme, aborting on the first encountered conflict. Transactions that are more complex switch dynamically to the greedy mechanism that involves more overhead but favors these transactions, preventing starvation. Additionally, transactions that abort due to write/write conflicts back-off for a period proportional to the number of their successive aborts, hence reducing contention on memory hot spots.

Again, this very brief summary of the distinctive features of SwissTM is included here only to show why it has become a major player among the various STMs used nowadays by the research community.

# 3 Benchmarks

In this section we provide a summary of the benchmarks that are going to be explored during the project. STAMP [10] has been chosen because it is one of the most used benchmarks in the literature. STMBench7 [5] is here because it is based on a very interesting concurent data structure that allows fine control over the variables that affect the performance of transacional memories. eigenBench [7] is included due to its flexibility.

## 3.1 STAMP

The STAMP benchmark [10] was the first to include a series of wide range real applications tailored test transactional memory implementations. The applications included in the benchmark are: bayes, genome, intruder, kmeans, labyrinth, ssca2, vacation, and yada. Due to space limitation, we comment on four of the eight applications.

- bayes: it implements and algorithm for learning the structure of Bayesian networks from observed data. The algorithm uses a hill-climbing strategy that combines local and global search.
- genome: genome assembly is the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. This is a two-phase algorithm and transactions are used in each phase to organize the concurrency upon the hash table the stored the genome segments.
- intruder: This is a network packet scanner, that is, it filters network packets basd on a set of intrusion signatures, using techniques developed for network intrusion detection systems. Network packets are processed in parallel and go through three phases: capture, reassembly, and detection. The main shared data structure are a queue and a dictionary (hash) whose methods are organized as transactions.
- labyrinth: this program implements a classical routing through a maze algorithm that was the basis in the past for printed-circuit board routing. The main data structure is a three-dimensional uniform grid that represents the maze. Transactions are used to organize parallel searches through this maze. Labyrinth tends to spend most of its computation time on calculations to find shortest distances over the grid, thus it contains long duration transactions with high levels of contention upon the shared data structure.

A detailed discussion of the parameters can be found in [10].

Table 1: Summary of application characteristics included in STAMP in terms of transaction parameters

| Application | Tx Length | R/W set | Tx Time | Contention |
|---|---|---|---|---|
| bayes | long | large | high | high |
| genome | medium | medium | high | low |
| intruder | short | medium | medium | high |
| kmeans | short | small | low | low |
| labyrinth | long | large | high | high |
| ssca2 | short | small | low | low |
| vacation | medium | medium | high | low/medium |
| yada | long | large | high | medium |

### 3.2 STMBench7

STMBench7 [5] and eigenBench [7] differ from STAMP [10] in the following aspects. STMBench7 is not based on different applications but on a shared tree (graph) data structure dynamically generated as a function of initial parameters that determine the size, breadth, depth, and balance of the tree. Transactions with different characteristics in terms of length, read-write data sets, time and contention are them executed upon the tree to emulate different profiles of transactional applications.

### 3.3 eigenBench

eigenBench has been developed by the same group that developed STAMP at Stanford University. Their experience with STAMP has led the group to isolate eight parameters that they consider necessary and sufficient to model any transactional application. The parameters are:

- concurrency: number of concurrently running threads;
- working-set size: size of frequently used shared memory;
- transaction length: number of shared accesses per transaction (ratio of reads to writes that should be conditioned by transactions);
- pollution: fraction of shared writes to shared accesses;
- temporal locality: probability of repeated address access per shared access;
- contention: probability of conflict of a transation against any other active transaction;
- predominance: fraction of shared access cycles to total execution cycles;
- density: fraction of non-shared cycles executed outside the transactions to total non-shared cycles.

As we can see from the summaries above, the three benchmarks selected represent a carefully chosen cross cut of the benchmarks avaliable. STAMP is based on the straight implementation of applications, STMBench7 is a benchmark based on a single shared data structure that is accessed/updated using transactions. eigenBench can be seen as a application emulator as it has isolated eight orthogonal characteristics of transaction-based applications that can be combined to emulate any reasonably justifiable transactional application (benchmark).

# 4 Operating System: external factors

From the perspective of the transactional memory implementation the operating system and the hardware are the environment. In this section we summarize the factors whose change is going be used to assess the change in the speed-up of the transactional memory.

We have chosen Linux as the operating system for the experiments for the simple reason that it is by far the dominant operating system used for research on transactional memory. Further details about each of the Linux components mentioned here can be found by reading the kernel code with the help of text such as the one written by Bovet and Cesati [1], or Love [9].

## 4.1 Scheduler (Process Manager)

Linux, as any time sharing system, achieves concurrent programming, be it in unicore or multicore hardware, by switching the CPU from one process (thread) to another in a very short time frame. The data structures and the actual event of process switch is not going to be detailed in this text. Here we are concerned only with the mechanisms used by the kernel to decide when and which process to switch in and out of a CPU; this is called *process scheduling*.

From the point of view of a TM, any change in the behaviour of the scheduler has a potential to change its speed-up (performance) and it is interesting to see how changes in, for example, the scheduling policy, affect the benchmark.

Linux scheduling is based on a time sharing mechanism: several threads run in "time multiplexing" because the CPU (or core) time is divided into slices, one for each runnable process. If a currently running process has still not finished its execution when its time slice (quantum) expires, a process switch may take place. Time sharing relies on timer interrupts and is thus transparent to processes. Process switching is based on a mechanism that dinamically assigns *priorities* to processes. The scheduler keeps track of what processes are doing and adjusts their priorities periodically to maintain a balanced used of the CPU by processes that have

different characteristics in terms of resources use (CPU, I/O, memory, etc).

In the current version of the Linux kernel, a process is scheduled according to one of the following scheduling classes: fifo, real time, and normal. A process belonging to each of these classes can have its static priority, dynamic priority, quantum, and core affinity changed. There are several system calls that allow the conditioning of the behaviour of the scheduler, which are not detailed here due to space limitation. We are going to devise experiments to test whether changes in the behaviour of the scheduler can have a significative impact on the results resported by the benchmarks.

### 4.2 Memory Manager

We use to think of the computer's memory as a homogeneous, shared resource. Disregarding the role of the hardware caches, we expect the time required for a CPU (core) to access a local memory to be essentially the same, regardless of the physical location and the CPU (core). Unfortunately, this assumption is not true for some architectures. The kernel has a set of system calls that allows the tuning of a few parameters that affect the behaviour of the memory manager such as: caching, locality of access, page fault behaviour, and contention.

### 4.3 Threads Library

Linux allows threads to be executed basically in two ways: using a user-level threads library that allows many different mappings between user-level threads and kernel threads, or using a straight one-to-one mapping between user-level threads and kernel-level threads via their native threads implementation (NTPL). So, we would like to access the impact on the benchmarks of the choice of policy used to map user-level threads to kernel-level threads.

## 5 Methodology

The methodology to be used in the project has already become a standard for researchers that work in the field of transactional memory. A configuration consisting of a benchmark, a STM implementation and a multicore machine is setup—we have at our disposal machines with 16 and 32 cores. Then, the benchmark is run and the results come as graphs showing the speed-up obtained for a given benchmark application as the number of cores is scaled up. In this project the same procedure is going to be used with the additional step that first a Linux kernel is going to be setup with the parameters selected for the external factor under test. For example, if the external factor to be tested is a different scheduling algorithm, then the process scheduler is going to be setup

so that the desired scheduling algorithm is active during the experiment. Later, the comparison of the results obtained with a modified kernel against the results obtained with a kernel without changes are going to allow us to observe whether the factor under test had an impact in the behaviour of the benchmark. Statistical analysis of the results are going to be based upon standard sampling and variability analysis as described in chapter 13 of Jain's text [8].

## 5.1 Schedule

This section details the tasks planed for the project and their respective time schedule. (Table 2):

1. Tansactional memory: study of fundamental concepts.
2. eigenBench: parameters, compilation, STM binding, execution, data collection, result analysis.
3. STAMP: parameters, compilation, STM binding, execution, data collection, result analysis.
4. STMBench7: parameters, compilation, STM binding, execution, data collection, result analysis.
5. Statistics fundamentals with R.
6. Linux: process management, scheduler, threads.
7. TinySTM: study of its parameters and behaviour.
8. SwissTM: study of its parameters and behaviour.
9. Experiments with STAMP, STMBench7, eigenBench, TinySTM: process manager, memory manager, threads (Linux).
10. Experiments with STAMP, STMBench7, eigenBench, SwissTM: process manager, memory manager, threads (Linux).
11. Writing of a technical report/paper.

Table 2: Project Schedule

| | 2013 | | | | 2014 | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tasks | aug | sep | oct | nov | dec | jan | feb | mar | apr | may | jun | jul |
| 1 | ■ | ■ | ■ | ■ | ■ | | | | | | | |
| 2 | | ■ | ■ | ■ | ■ | | | | | | | |
| 3 | | ■ | ■ | ■ | ■ | | | | | | | |
| 4 | | ■ | ■ | ■ | ■ | | | | | | | |
| 5 | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | | |
| 6 | | ■ | ■ | ■ | | | | | | | | |
| 7 | | ■ | ■ | ■ | | | | | | | | |
| 8 | | | | | ■ | ■ | ■ | ■ | ■ | ■ | | |
| 9 | | | | | ■ | ■ | ■ | ■ | ■ | ■ | | |
| 10 | | | | | ■ | ■ | ■ | ■ | ■ | ■ | | |
| 11 | | | | | | | | ■ | ■ | ■ | ■ | ■ |

# References


[1] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly Media, 2008.

[2] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[3] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[4] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[5] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *ACM SIGOPS Operating Systems Review*, 41(3):315–324, 2007.

[6] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[7] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11, 2010.

[8] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[9] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[10] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35 –46, sept. 2008.