Locking and Transational Memory: A Performance Comparison

Felipe Ebert[a], Weslley Torres[a]

[a]*Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Recife, Brazil*

Abstract

The main goal of this paper is to present the concepts and compare the performance of the following concurrency control mechanisms: locking and transactional memory. This work focuses on the C programming language. We present a performance comparison between transactional memory types and performance comparison between software transactional memory and coarse-grain and fine-grain locks. As result, we could observe that locking has a better performance than software transactional memory when a few number of threads is used.

Keywords: Locking, Transactional Memory, Performance, Benchmarks.

1. Introduction

Parallelism has gained a lot of importance in the past few years, even small systems are taking the advantage of synchronization that is now the mainstream. On one hand, locking has a history of successful use in production but its shortcomings are also well-known. On the other hand, Transactional Memory (TM) has been viewed as a mechanism of synchronization. However, TM is also known by its advantages and disadvantages, but these shortcomings seems to be less understood than locking. In fact, it seems that locking is still dominating the industry.

In this paper we present the characteristics and advantages/disadvantages of locking and TM. The focus is in the comparison of their performance. We firstly quickly compare their performance individually, then we compare five microbenchmarks of locking and a specific type of TM.

The reminder of this paper is organized as follows. The background is presented in Section 2. Section 3 discuss about the characteristics of each synchronization mechanism and their advantages/disadvantages. In Section 4 we present the performance discussion comparing locking and TM. And finally, Section 5 concludes this work.

2. Background

In this section we introduce transactional memory in Section 2.1 and locking in Section 2.2.

2.1. Understanding Transactional Memory

TM provides a flexible and simple mechanism for parallel programming on multicore processors. Unlike traditional control mechanisms of synchronization such as locks, TM uses a high-level abstraction of concurrency control. Thus, the programmer does not have to worry about inherent problems of concurrent programming. For example, data synchronization is a responsibility of the TM system, which becomes responsible for ensuring the correct operation of the application, avoiding the main problems (deadlocks, race conditions) when programmer uses mechanisms of locks.

TM is a way of concurrency control to manager the access to shared data in concurrent systems. It resembles databases' transactions which according to Herlihy et al. [19] a transaction is a sequence of steps executed by a single thread. Transactions must be serializable, i.e. they appear to execute sequentially, in a one-at-time order. According to Lintzmayer et al. [21], a transaction is a piece of code that performs a sequence of reading and writing operations that occur in a single instant of time, and whose state is not visible to others threads until the operation is not totally completed.

According to Elmasri and Navathe [10], the concept of transaction in the database system is that a transaction is a running program or process that includes one or more accesses to the database, such as reading or updating of a data in a database. Another definition is given by Harris et al. [15], which says that a transaction is a sequence of actions that appear indivisible to an outside observer. Thus, there is only two possible ends to a transaction, the first one is when the operation went fine and there were no problem so the operation made the commit. The second one when there were problems and the operation had to rollback and abort the operation.

The concept of transaction memory is inherited from the concept of database transactions. As transactions in database systems, TM have properties to ensure the transactions' context will not be lost or mixed with others transactions . A transaction in database systems has these four properties: atomicity, consistency, isolation and durability [4]. Most of TM implementations do not provide the consistency and durability properties. The former due to the fact that memory are volatile, the latter is not applied because TM do not have the concept of metadata,

with no consistency rules to be guaranteed during the transaction. Therefore, the following properties are valid for TM:

- **Atomicity**: refers to the ability of the system to ensure that all tasks of a transaction are executed (committed) or none are;
- **Isolation**: refers to the ability of the system to ensure that a transaction in progress (not yet committed) can not be accessed by any other transaction;

Below we describe briefly how the TM works [12, 29]:

- The transaction's result is not visible to others transactions until the transaction commit;
- Each transaction is executed in a single thread without acquiring lock;
- When a transaction starts, the system keeps the old values, which can be restored if aborted;
- A transaction can not write directly in shared memory. Instead, the result is stored in an undo-log or a write-buffer;
- If the transaction complete without any problem, then the transaction commits and the result is stored in the shared memory. Otherwise, if a conflict is detected between two or more transactions, only one will be committed while the others will rollback i.e restore the previously saved data

TM can be classified into two main categories: Software Transactional Memory (STM) [27, 28] and Hardware Transactional Memory (HTM) [3, 18]. The former, which will be addressed in this paper, are implemented using compilers and libraries used to control the transactional execution. The latter, hardware transactional memory, the implementation of the transactional monitoring were implemented directly in the hardware through changes in the cache, architecture, bus and coherence protocols. Besides these two main categories there is also the possibility of a hybrid implementation of transaction memory (HyTM) [6, 25], ie, an implementation that supports the co-existence of transactions in hardware and software at the same time thus it would be possible to get the best of each.

Here we can list some advantages and disadvantages of using STM [29] instead of HTM:

Advantages:

- STM is the easiest to be implemented, modified and has few architectural limitations compared to the hardware;
- STM Requires no changes to existing hardware;
- STM has application portability without the need for specific hardware;
- In STM there is no size transaction limit, on the other hand, there is a limitation on size transaction in HTM.

Disadvantages:

- Most STMs have poor performance rate;
- Weak atomocity, which may produce incorrect or unpredictable even for simple parallel programs that wold work correctly with lock-based synchronization [8, 20].

According to Harris et al. [16] and Tabatabai [1] even though the code can be optimized by compilers, STM can still slow down each thread by 40% or more.

2.1.1. Software Transactional Memory

To the best to our knowledge, the first work of TM focusing exclusively on software was [28]. Since then, several papers presented new software-based and new mechanisms to minimize the overhead generated by STM systems. We present some of the main STM systems.

RSTM. It was developed by researchers of the University of Rochester in 2006 [22]. Early versions of the system were object-based and obstruction-free. As described by Herlihy et al. [17], an obstruction-free algorithm guarantees that a given thread, starting from any feasible system state, will make progress in a bounded number of steps if other threads refrain from performing conflicting operations. The recent versions of the system supports multiple algorithms, approximately 13 word-based and lock-based algorithms.

TinySTM. It is an open-source word-based and lock-based STM system [11]. It relies upon a shared array of locks to manage concurrent accesses to memory. Each lock covers a portion of the address space and uses a per-stripe mapping where addresses are mapped to locks based on a hash function The TinySTM system was proposed in 2008 and has its latest version November 2011. TinySTM compiles and runs on 32 or 64-bit architectures.

Intel STM. It consists of a C/C++ STM library and a runtime compiler for Linux or Windows producing 32 or 64 bit code for Intel or AMD processors[1]. The compiler interacts with shared memory within transactions through barriers. Intel STM implements atomic keyword as well as providing ways to decorate (declspec) function definitions to control/authorize use in atomic sections. Intel STM generates transaction statistics that can help the user understand the application performance in order to find bottlenecks. Some of the information are: the number of times a transaction were performed, the number of failed transaction executions due memory conflict.

2.2. Understanding Locks

A lock is a synchronization mechanism in multithread environments which limits access to specific resources. Locks are used to protect specific parts of the source code. If a thread holds a lock for a specific part of the source code, then any other thread can execute that piece of code until the fist thread release the lock. It is the mutual exclusion concurrency protocol.

There are several types of locks. In this section we will discuss only few types of locking [23].

Exclusive Lock. This lock guarantees that only one thread holds the lock at a time. It means the lock's holder has exclusive read and write access to the protected data.

Read-Writer Lock. This lock permits only one writer thread to hold the lock at a time or any number of readers threads to hold the lock. It has an excellent scalability for data that is read often and written rarely.

Scoped Locking. This lock does not need explicit primitives to acquire or release the lock. It is based on the object-oriented *resource allocation is initialization* pattern [9] in which the corresponding constructor is invoked upon entry to the scope of the object, and the destructor is invoked upon exit from that scope. In this lock the constructor acquires the lock and the destructor releases it.

The most simple primitive of locking in C/C++ is *mutex* [5, 2, 26]. It provides mutual exclusion by simply representing an integer in memory, and as presented

[1] `https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`

the types of locks before, mutex is an example of *exclusive lock*. Its initial value is 0 which means it is unlocked. The first operation defined on a mutex is called *lock* which tests the state of the lock, and then locks the mutex (change its value to 1) if it is unlocked (if the value is 0). The second operation is called *unlock* which unlocks the mutex (changes its value to 0).

The problem is when two threads when trying to acquire the lock read 0 at the same time, then both would write 1 and think they got the lock. The trick here is that the test and set operation are atomic operations. Luckily CPUs has functions like "compare-and-set" and "test-and-set" which does such atomic operation, they can perform compare/test and set the value (in case the test was successful) as there were a lock for them.

Furthermore, every mutex has a list of threads that are waiting for it to unlock. Whenever a thread try to lock and it is not possible (because the mutex is already locked by another thread) then it adds itself to the waiting list of that mutex. When the mutex is unlocked, then the OS system picks one threads from the waiting list and wakes it up. This thread will try to lock the mutex again then. Of course the simplicity of mutex has its disadvantages as we will discuss later in this paper.

3. Comparing Characteristics

In the following sections, we will present some advantages and disadvantages between locks and TM. The advantages and disadvantages presented here were listed by McKenney et al. [24].

3.1. Advantages of Transactional Memory

TM is a new and promising field, but it is already possible to list a number of advantages:

- Does not cause deadlock;
- The fact that transactions are atomic make it easy to program and understand multi-threaded code;
- Some TM implementations are non-blocking, thus, any delay or failure in a thread not hinder the progress of other threads. This kind of fault tolerance is extremely difficult to obtain when using locking;

3.2. Disadvantages of Transactional Memory

Some of the disadvantages of TM are:

- Rollbacks can result starvation of large transactions by smaller ones and;
- Delay of high-priority processes via rollback of the high-priority process's transactions due to conflicts with those of a lower-priority process;
- STM does not have a good performance compared to that of locking [22];
- There is a poor interaction between TM and existing software tools. For example, in many HTM proposals, the traps induced by breakpoints can result in unconditional aborting of enclosing transactions, reducing this common debugging technique to an exercise in futility;

3.3. Advantages of Locking

The first advantage of locking is its pervasiveness, it is widely used. Below we describe some advantages of using locking [24]:

- There is an huge amount of experienced developers, therefore there are a huge number of concurrent systems developed with locks;
- Locks can be used on existing commodity hardware. There are well-defined and standardized locking APIs, it allows the concurrent code based on locks to be run on several platforms. Furthermore, lock minimally harms the system's performance, there are even CPUs with special instructions to handle locking;
- Locks interacts well with huge number of synchronization mechanisms, which includes atomic operations, non-blocking synchronization, reference counting, and read-copy update (RCU);
- The contention effects are concentrated within locking primitives. As result the critical section can run at full speed. Different from others mechanisms that contention degrades critical section performance.

3.4. Disadvantages of Locking

Some of the disadvantages of locking are:

- **Deadlock**: when systems use more than one lock it is possible to them to deadlock. It happens when one thread has a lock and attempts to acquire another lock held by another thread, while that another thread is attempting to acquire the lock held by the first thread;
- **Priority inversion**: it occurs when a lower-priority thread is preempted while holding a lock needed by higher-priority threads.
- **Blocking on thread failure**: whenever a threads terminates while holding a lock, any other thread trying to acquire the lock will be blocked indefinitely;

Other disadvantages are the high synchronization overhead that lock imposes in certain situations, the high contention on non-partitionable data structures and also the non-deterministic lock-acquisition latency which can be an issue for real-time workloads.

4. Comparing Performance

In this section we will present some performance tests both using lock and transactional memory. Then we will discuss five microbenchmarks comparing both mechanisms.

4.1. Transactional Memory Performance

In this section we present some performance results between three STM implementations: RSTM, TinySTM and SwissTM. This result can be found in [14].

Public benchmarks were used in order to assure the fairness of the test and make it easy to the academic community repeat it. The following benchmarks were chosen to perform the tests:

- EigenBench - It is a simple synthetic benchmark for TM systems. In this benchmark there are the following eight orthogonal TM characteristics: concurrency, transaction length, working set size, temporal locality, pollution, contention, predominance, and density. EigenBench allows the users to explore each of these TM characteristic, independently.
- STAMP - This benchmark has eight different domain applications. It is widely used by academic community.

Below we present the results for speedup tests using the STAMP benchmark. Figure 1 presents the overall speedup results of systems with all benchmark application.

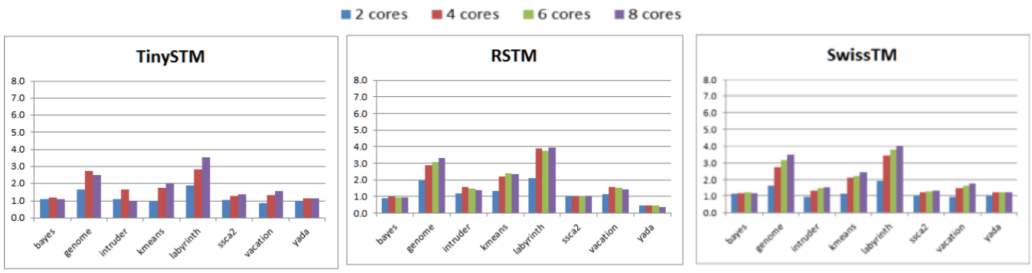

Figure 1: Speed-up of STM systems with the STAMP benchmark [14].

Figure 1 presents that RSTM and SwissTM had greater speedup than TinySTM. According to Furlan Rui [14] this finding was expected due to frequent updates on RSTM and SwissTM which demonstrates the effort to gain better results, as indeed we observed in Figure 1. It was also possible to observe scalability problems when six and eight cores were used. The results indicate that the systems still have difficulties in scale up of four cores. Additionally, all systems have problems running SSCA2, Bayes and Yada applications due scalability problems. Finally, Intruder, Vacation and Kmeans applications despite having superior results to other applications they presented low speed-ups.

Furlan Rui could not explain the reasons for the low performance, but he explained the fact that some applications are not fully transactional so they might perform poorly. These applications are not 100% prepared to run in a Transactional way so it would not be possible to gather ideal speedup values.

Among the eight applications, only Genome and Labyrinth achieved a good speedup. The Labyrinth application presents great performance with two cores on all STM. However with four cores, only RSTM and SwissTM had a good performance. Genome showed optimal performance with two cores in RSTM. Both SwissTM and RSTM had similar results, above four cores they perform best.

Below we present the results for speedup tests using the EigenBench benchmark. Figure 2 presents the overall speedup results. According to Furlan Rui [14], the purpose of this test is to use the same applications used in the previous test and perform a more complete analysis about the performance of selected STM. SwissTM and RSTM were selected to perform this test and one of the reasons that

these STM had been chosen was due the similar results presented in the previous test.

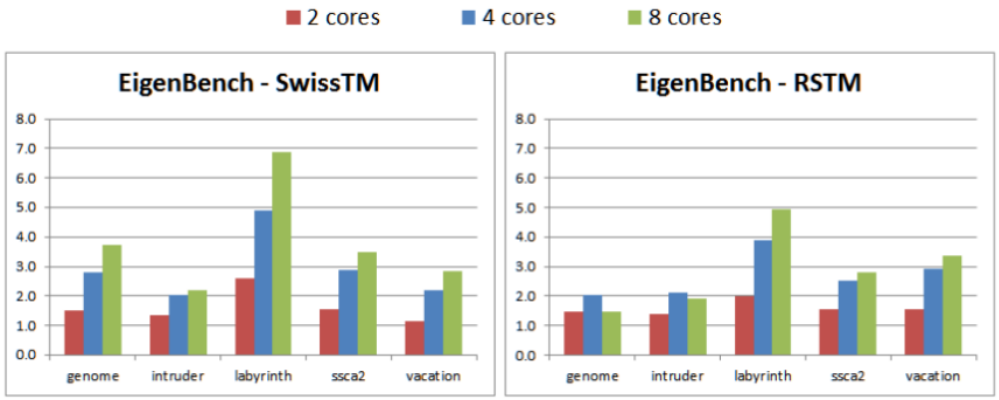

Figure 2: Speed-up of STM systems with the EigenBench benchmark [14].

Five applications used in the previous test were also used in this one, below we describe the selected application:

- SSCA2: Application that had low performance.
- Intruder and Vacation: Applications that had an average performance.
- Labyrinth and Genome: Applications that have high performance.

In this test, the number of aborts per Commit (AFT) were taken a count, AFT represents the percentage of transactions that had to abort divided by the number of committed transactions.

In Figure 2 we can observe the superiority of SwissTM. It had a better performance in 4 of 5 applications. SwissTM had a better performance with 8 cores due to its sophisticated mechanism for conflict resolution, which prevents long transactions abort more often as can be seen in Figure 3. The Genome application, showed a significant difference between the STM, which speed-up with eight cores was 3.7 and 1.4, for SwissTM and RSTM respectively.

This huge difference between these STMs is due the behavior of the Genome application, which shows variation between lengths of short and long transactions and is best worked in SwissTM due it was made to work with both short and long transactions

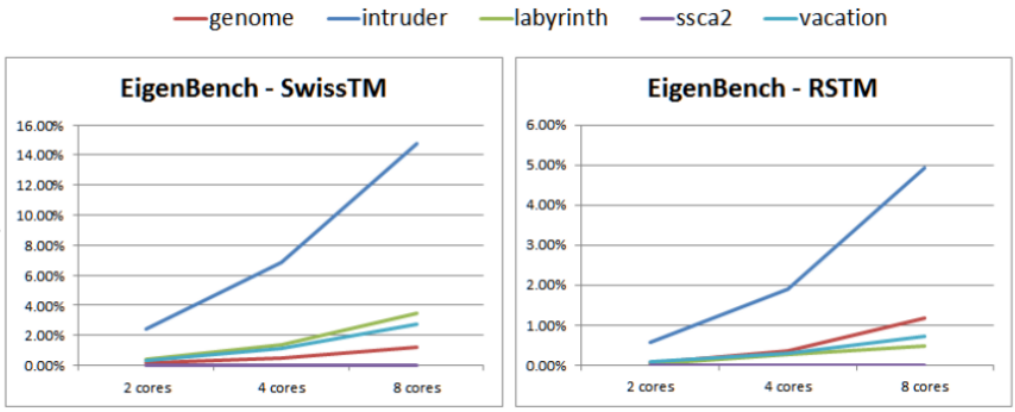

Figure 3: ApC (Aborts per Commit) benchmark EigenBench [14].

4.2. Locking Performance

We will start discussing the locking performance presenting the results by [13]. He compared the performance of three different cases of locking in C++: mutex, spin lock and no locking. The benchmark created by [13] is explained below.

There is a counter for each case which increments a volatile value as shown in Figure 4:

```
volatile int value = 0;

int mutexLock(int limit) {
        for (int i = 0; i < limit; ++i) {
                std::lock_guard<std::mutex> guard(mutex);
                ++value;
        }
        return 0;
}

int spinLock(int limit) {
        for (int i = 0; i < limit; ++i) {
                spinlock.lock();
                ++value;
                spinlock.unlock();
        }
        return 0;
}

int noLock(int limit) {
        for (int i = 0; i < limit; ++i) {
                ++value;
        }
        return 0;
}
```

Figure 4: The three counters: mutex, spin lock and no locking.

Then he called each method with the limit of 10000000. The results of the benchmark are presented in Figure 5. As we can see, when there is no lock the counter's time is 58 ms, which is much faster than the counters with lock but the counter's value at the end may present some garbage, as expected. And the spin lock (1589 ms) is almost 2500x faster then mutex (38942 ms). However, the spin lock has the disadvantage of consuming the CPU cycles while the lock is spinning.

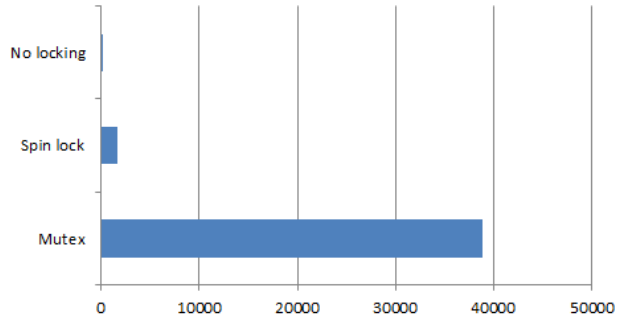

Figure 5: The results of the benchmark by [13] in milliseconds.

Another similar locking benchmark was run by [7]. He compared the performance of five different cases of locking in C++: no locking, lock, atomic, spin lock and mutex. He used the same idea of counting. The results of this benchmark is presented in Figure 6.

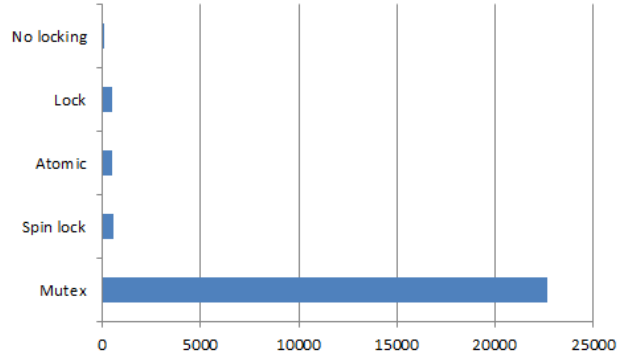

Figure 6: The results of the benchmark by [7] in milliseconds.

Here again the fastest is no locking, however there is garbage in the counter at the end. Lock and atomic have similar times because, under cover, atomic also uses lock in a portable and guaranteed way. Spin lock is just a little slower than lock and atomic due the spinning time. And mutex over again is the slowest.

4.3. Transactional Memory vs. Locking Performance

We will present now the comparison between TM (RSTM) and locking (coarse and fined grained locks) made by [22]. The RSTM was tested with each combination of visible/invisible reads and eager/lazy acquire. They used five microbenchmarks:

LinkedList. It is a sorted linked list and three operations are executed: insert, delete and lookup. It traverses the nodes in the list in read-only mode, then when the target node is found it uses read-write access.

HashTable. It consists of 256 buckets with overflow chains. The tests perform roughly equal numbers of insert and delete operations, so the table is about 50% full most of the time.

RBTree. It is a red-black tree which uses node values in the range 0..65535. The search down the tree is made in read-only mode. When the target node is

found, it uses read-write access and goes back up the tree opening the nodes that are relevant to the height balancing process also in read-write mode.

RandomGraph. The graph is implemented as a sorted adjacency list. Each newly inserted vertex initially receives up to four randomly selected neighbors. The search is done in read-only mode and open the target node in read-write mode. Furthermore, the transaction looks up each neighbor of the target node which was affected and then updates them.

LFUCache. It is a web cache simulation using least-frequently-used page replacement. It uses a large (2048-entry) array-based index and a smaller (255-entry) priority queue to track the most frequently accessed pages in a simulated web cache. When re-heapifying the queue, it always swap a value-one node with any value-one child, this induces hysteresis and gives a page a chance to accumulate cache hits.

Now we present the results of performance comparison tests between TM and locks. The y axis in each Figure plots transactions per second on a log scale. With the small number of threads, the coarse-grained lock (CGL) had a better performance on all benchmarks tested. The difference in performance varied between 2 times higher as shown in Figure 7 and up to 20 times higher as presented in Figure 8.

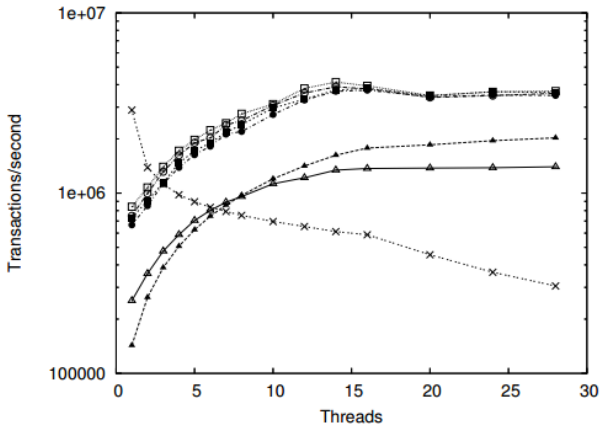

Figure 7: Hashtable benchmark [24].

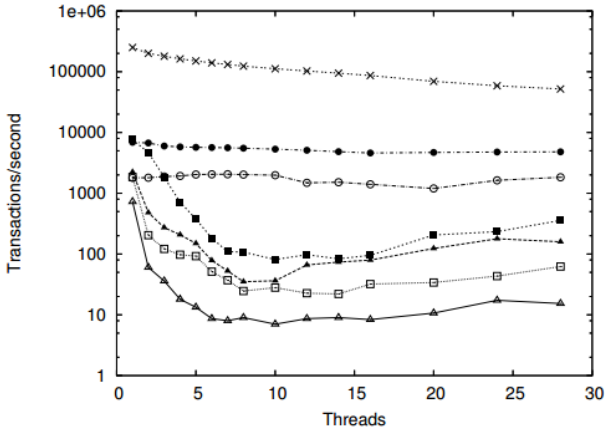

Figure 8: RandomGraph benchmark [24].

However, as the number of threads increases RSTM overtakes CGL in benchmarks that permit concurrency. This performance improvement can be noticed in the Figure 7, which shows that with three threads the GCL performance falls dramatically while RSTM performance increased. In RBTree benchmark RSTM overtakes CGL when the number of threads is 14 as presented in Figure 9, and in Linkedlist benchmark RSTM needs 20 threads to overtakes CGL as presented in Figure 10. In the LFUCache and RandomGraph benchmarks, Figures 8 and 11 respectively, neither of which admit any real concurrency among transactions, CGL is always faster than transactional memory.

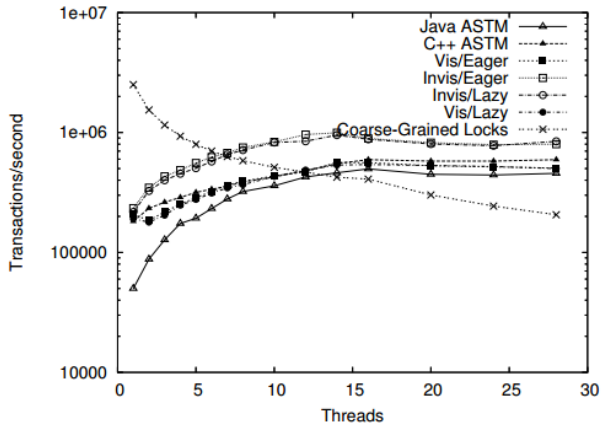

Figure 9: RBTree benchmark [24].

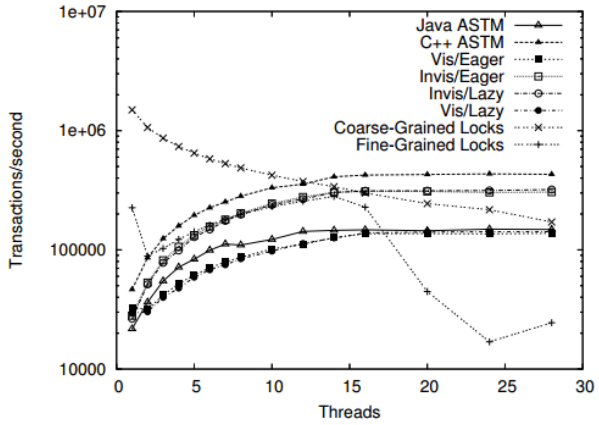

Figure 10: Linkedlist benchmark [24].

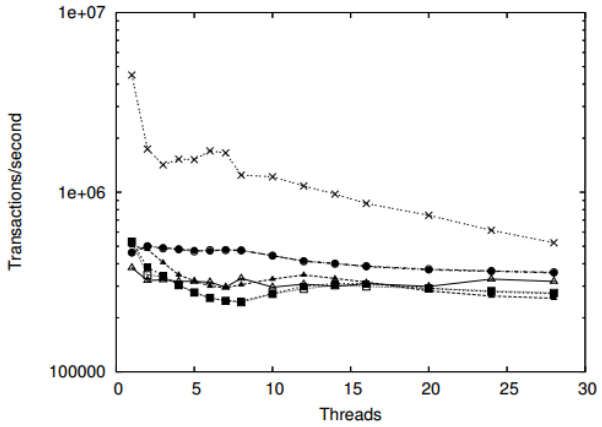

Figure 11: LFUCache benchmark [24].

In order to assess the benefit of early release, Marathe et al. [22] compared the LinkedList benchmark to a "hand-over-hand" fine-grain locking (FGL) implementation in which each list node has a private lock that a thread must acquire in order to access the node, and in which threads release previously-acquired locks as they advance through the list. Figure 10 shows that FGL has a good performance however when the number of thread reaches 15 the performance drop dramatically and the FGT becomes the worst implementation.

5. Conclusions

This paper is a literature review about TM and lock implementation in C and C++. This paper also presents a performance comparisons between TM types, performance comparison between STM and Coarse-Grain and Fine-Grain Locks. We can conclude that although the HTM present a better performance than STM, HTM contains more critical limitations for example, the need to hardware modification, limited transaction size.

We also present the results obtained by Furlan Rui [14] on the comparison between three STM (RSTM, TinySTM and SwissTM). Furlan Rui showed that SwissTM had the best performance of the tested STM but Furlan Rui believes that much still needs to be studied to improve performance, especially scalability issue, because STM lose performance as the number of cores increases.

Marathe et al. [22] presented the performance comparison test between STM and Lock and conclude that lock has a better performance than STM when a few number of threads is used. As the number of threads increases, the STM performance improves.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. *SIGPLAN Not.*, 41(6):26–37, June 2006.

[2] Himanshu Arora. How to use c mutex lock examples for linux thread synchronization. Technical report, Address: http://www.thegeekstuff.com/2012/05/c-mutex-examples/ – Last access: Sep 5th 2014, May 2012.

[3] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 24–34, New York, NY, USA, 2007. ACM.

[4] Mike Chapple. The acid model.

[5] Cplusplus. Lock mutex. Technical report, Address: http://www.cplusplus.com/reference/mutex/mutex/lock/ – Last access: Sep 5th 2014.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, October 2006.

[7] Alexander Demin. Comparing the performance of atomic, spinlock and mutex. Technical report, http://demin.ws/blog/english/2012/05/05/atomic-spinlock-mutex/, May 2012.

[8] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006. Electronic, no. page numbers.

[9] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[10] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (2Nd Ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[11] 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.

[12] Cesare Ferri, Tali Moreshet, R. Iris Bahar, Luca Benini, and Maurice Herlihy. A hardware/software framework for supporting transactional memory in a mpsoc environment. *SIGARCH Comput. Archit. News*, 35(1):47–54, March 2007.

[13] Tom Fewster. Compare c++ locking mechanisms for performance. Technical report, http://www.wannabegeek.com/?p=557, September 2013.

[14] Fernando Furlan Rui. Uma avaliação comparativa de sistemas de memória transacional de software e seus benchmarks. 2012.

[15] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[16] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 14–25, New York, NY, USA, 2006. ACM.

[17] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[20] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.

[21] Carla Négri Lintzmayer, Eduardo Theodoro, and Maycon Sambinelli. Memória transacional.

[22] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *DEPT. OF COMPUTER SCIENCE, UNIV. OF ROCHESTER*, 2006.

[23] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2010.

[24] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: A comparison of locking vs. transactional memory. *SIGOPS Oper. Syst. Rev.*, 44(3):93–101, August 2010.

[25] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun.

An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 69–80. ACM Press, 2007.

[26] Mortoray. How to use c mutex lock examples for linux thread synchronization. Technical report, Address: How does a mutex work? What does it cost? – Last access: Sep 5th 2014, Dec 2011.

[27] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[28] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[29] C. Fu X. Wang, Z. Ji and M. Hu. A review of transactional memory in multicore processors. *Information Technology*, pages 192–200, 2010.