2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing

Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures

Márcio Castro*, Kiril Georgiev†, Vania Marangozova-Martin*, Jean-François Méhaut*,
Luiz Gustavo Fernandes§ and Miguel Santana†
*MESCAL Research Group (INRIA - LIG - Grenoble University), France
Email: FirstName.LastName@imag.fr
§GMAP Research Group (PPGCC - PUCRS), Brazil
Email: luiz.fernandes@pucrs.br
‡STMicroelectronics, Crolles, France
Email: Kiril.Georgiev@st.com, Miguel.Santana@st.com

***Abstract*—Transactional Memory (TM) is a new programming paradigm that offers an alternative to traditional lock-based concurrency mechanisms. It offers a higher-level programming interface and promises to greatly simplify the development of correct concurrent applications on multicore architectures. However, simplicity often comes with an important performance deterioration and given the variety of TM implementations it is still a challenge to know what kind of applications can really take advantage of TM. In order to gain some insight on these issues, helping developers to understand and improve the performance of TM applications, we propose a generic approach for collecting and tracing relevant information about transactions. Our solution can be applied to different Software Transactional Memory (STM) libraries and applications as it does not modify neither the target application nor the STM library source codes. We show that the collected information can be helpful in order to comprehend the performance of TM applications.**

***Keywords*-software transactional memory; tracing mechanism; benchmark.**

I. Introduction

Multicore technology proves to be a promising solution to the problem of achieving higher performance without increasing power consumption. Because of that, it is strongly expected that the number of processor cores will continue to increase, resulting in manycore architectures with hundreds or even thousands of cores. In this context, the development of applications with high degrees of parallelism and the correct management of complex synchronization issues become a major concern.

Traditional synchronization structures such as *locks*, *mutexes* and *semaphores* are extensively used in a multicore context. They are simple to implement in hardware and they offer a safe solution to the problem of multiple threads sharing data. However, they have several disadvantages: (i) they are "low-level" mechanisms, since one must explicitly control the access to shared variables; (ii) they cause blocking, so threads always have to wait until a lock (or a set of locks) is released; (iii) they are hard to manage effectively, especially in large systems; and (iv) they can be vulnerable to failures and faults, such as deadlocks and livelocks.

Transactional Memory (TM) [1] has recently been proposed as an alternative synchronization solution. The idea is to offer a high level synchronization interface where developers only need to enclose concurrent accesses to shared variables in atomic sections (*transactions*). Problems such as correct synchronization, correct data race handling and deadlocks avoidance are shifted to the TM mechanism, which handles conflicts in an optimistic way [2].

Although TM promises to substantially simplify the development of correct concurrent programs, programmers will still need to debug code and study ways to optimize TM applications. It is clear that even with TM it is still a challenge to design and implement scalable concurrent programs. In this context, the questions we are interested in are the following. How can one know if an application will perform well with TM? How can one get useful details about the execution of TM applications? How can we use them to improve performance?

In this paper, we show that the performances of applications using TM-based synchronization solutions depend on both applications and TM solutions specifics. We demonstrate that, depending on these specifics, the use of TM may result in worse, equal or better performance for the application. In order to gain some insight on these issues, helping developers to understand and improve their performance, we propose an approach for collecting and tracing relevant information about transactions. Our solution can be applied to different STM libraries and applications as it does not modify neither the target application nor the STM library source codes.

The rest of this paper is organized as follows. In Section II, we describe the basic idea behind TM along with some important design criteria that impact TM performance. Sections III and IV motivate the use of STM as well as the necessity of tools to better comprehend TM applications. In Section V, we show our approach for tracing transactions.

1066-6192/11 $26.00 © 2011 IEEE
DOI 10.1109/PDP.2011.27

IEEE computer society

The collected information and results analysis are shown in Section VI. Section VII reviews some related works concerning STM. Finally, concluding remarks and future works are pointed out in Section VIII.

II. Transactional Memory

The basic idea behind *Transactional Memory* comes from transactional database management systems, in which a transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer. In the context of TM, a *transaction* is a portion of code that must be executed atomically and with isolation. A transaction may *commit* successfully, if its accesses to shared data did not conflict with other transactions; otherwise the transaction *aborts*, and none of its actions become visible to other threads. When a transaction aborts, the TM runtime *rollbacks* the conflicting transaction until it is possible to commit successfully.

Transactional Memory can be implemented in software (Software Transactional Memory) [3], [4], [5], in hardware (Hardware Transactional Memory) [6], [7] or in both (Hybrid Transactional Memory) [8], [9]. Software Transactional Memory (STM) has several advantages over Hardware Transactional Memory (HTM). It offers flexibility in implementing different mechanisms and conflict detection/resolution policies. It is easier to be modified or extended and is not limited by small fixed-size hardware structures, such as cache memories. Finally, STM does not require specific hardware, so it can be used on current platforms.

When designing a TM solution, four important criteria must be taken into account: transaction granularity, version management, conflict detection and conflict resolution.

- ***Transaction Granularity***: it defines the unit of storage for conflict detection [2]. For instance, in object-based languages, it is common to use *object granularity*, which detects conflicts when the states of shared objects are modified. Other examples are the *word granularity* and the *block granularity*, which respectively use memory words or groups of words for conflict detection. The transaction granularity cannot only have an important impact on the number of conflicts to be managed but also on the TM overall performance.
- ***Version Management***: since a transaction typically modifies data in memory, it is important to control how these modifications are managed on memory. There are two general ways to control it: *eager version management* and *lazy version management*. If the first one is applied, transactions will directly modify data and the system will use some sort of concurrency control to prevent other transactions from concurrently modifying objects. The system records the original data before updating, so it can be restored in case of transaction abortion. If *lazy version management* is used, transactions will deal with private copies of data. When a transaction commits, it updates the original data using the private copy.
- ***Conflict Detection***: there are two possible strategies to detect conflicts: *eager conflict detection* and *lazy conflict detection*. The first strategy detects read/write conflicts as they occur whereas the second one only detects at commit time.
- ***Conflict Resolution***: after detecting a conflict, the TM system must solve it. The usual solution is to abort one or more conflicting transactions. Usually, a TM system has a specific module called *contention manager*, which implements one or more *contention resolution policies* that are responsible for deciding which conflicting transaction must be aborted. The selected resolution policy clearly affects the performance of a TM system.

To sum up, TM solutions must take care of these issues in order to guarantee a functioning solution. The variety of possible combinations of such criteria clearly affects the behavior of TM applications as well as their performances.

III. STM versus Locks

The performance and benefits of using STM have been discussed since its first proposal in 1993 [10], [1]. In terms of performance, the research community tends to claim that it always results in considerably higher overheads than locks. However, this statement is not always true and it is not easy to foresee the performance of a TM application.

In this section we consider the well-known *Traveling Salesman Problem* (TSP) [11] in which the goal is to find the shortest possible path visiting each node of a graph exactly once. Here, we aim at comparing the performance of our two different solutions for the TSP: (i) using STM and (ii) using POSIX mutex locks.

In both TSP implementations the graph exploration is done by multiple threads which access shared variables managing the current shortest path and the pool of paths to explore. In the lock-based version, accesses to shared variables are enclosed by Pthread mutex lock/unlock sections. This is the case, for instance, of the accesses to the `minimum` variable, which stores the current shortest path (Listing 1).

```
1 void tsp(...) {
2    ...
3    pthread_mutex_lock(&mutex_minimum);
4    if (len < minimum)
5       minimum = len;
6    pthread_mutex_unlock(&mutex_minimum);
7    ...
8 }
```

Listing 1. Lock-based Accesses.

With STM, accesses to shared variables are enclosed by transactions whose boundaries are indicated by two special functions, namely `stm_start()` and `stm_commit()`. During a transaction, the shared variables are accessed using

stm_load() and stm_write() functions. If we consider again the example of the minimum variable, the above lock-based section is transformed in the following STM-based code. It should be noted that locks must be explicitly named by the programmer whereas this is not necessary when using transactions (Listing 2).

```
1 void tsp(...) {
2     ...
3     stm_start();
4     if (len < stm_load(minimum))
5        stm_store(minimum, len);
6     stm_commit();
7     ...
8 }
```

Listing 2. STM Access to Shared Data.

We have carried out several experiments on a Symmetric Multiprocessor machine (SMP) composed of four Intel Xeon X7460 (2.66GHz) processors with four cores each. This platform has 64GB of shared main memory and runs the x86_64 GNU/Linux operating system (kernel 2.6.262). The results were obtained through the average of 30 executions, presenting a low standard deviation.

We have used two approaches in implementing TSP. Our first approach is based on protecting *all* accesses to shared variables with the synchronization mechanisms. After a careful analysis, we noticed that some shared variables (such as minimum) have multiple read-only accesses allowing the removal of some "extra synchronization" without creating data races and then increasing even more the parallelism. This strategy was implemented in our second approach. The execution times of these two approaches are shown in Figures 1 and 2.

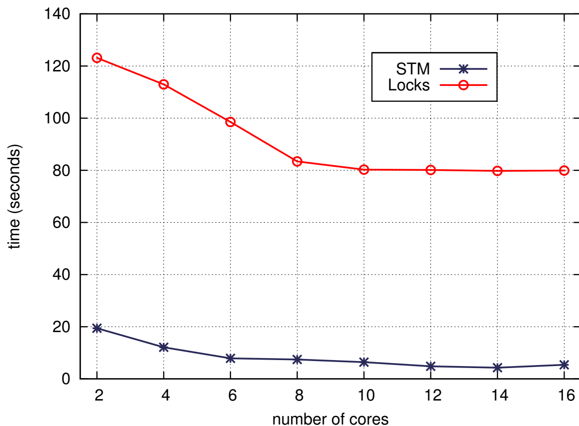

Figure 1. TSP Results: First Approach.

Considering the first approach, the results show a poor performance of TSP with locks in comparison to STM. This occurs due to the large number of accesses to the shared variable minimum, causing threads to be blocked continuously. On the other hand, it is not a bottleneck for the STM because of two reasons: (i) STM uses an optimistic approach to handle multiple accesses to shared data, so it does not block all threads; and (ii) most of the accesses to this shared variable do not conflict, which benefits such optimistic approach.

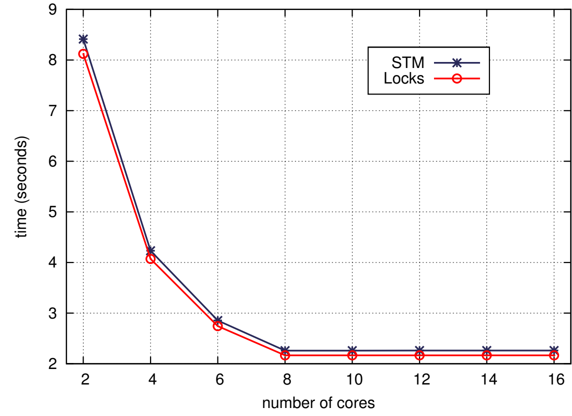

Figure 2. TSP Results: Second Approach.

On the other hand, it can be observed that the performance of the lock-based solution has drastically increased with the second approach. There are also some slight performance improvements for the STM solution. Both curves are very similar and no considerable overhead has been added by the STM (it is 5% worse on average), which shows that, depending on the applications characteristics, STM and locks can have similar performance.

IV. Performance Impact of STM Solutions

In order to investigate the impact of STM solutions on the performance of TM applications, we have carried out experiments with all non-trivial TM applications available from the *Stanford Transactional Applications for Multi-Processing* (STAMP). STAMP is a benchmark which includes 8 applications developed for TM [12]. However, in this paper we have selected the three most interesting case studies obtained from three different TM applications.

STAMP offers several advantages: (i) the applications use a variety of algorithms and belong to different application domains; (ii) it is possible to simulate different transactional behaviors varying the size of transactions (in terms of the number of instructions), the amount of contention and their granularity; and (iii) the applications can be easily executed with different STM libraries.

The selected applications from STAMP are the following:

- ***genome***: it takes a large number of DNA segments and tries to match them in order to reconstruct the original source genome. This application is characterized by medium-sized transactions and it spends most of its execution time executing transactions.

- ***intruder***: this application emulates a network intrusion detection system which scans network packets in order to detect a known set of intrusion signatures. This application is composed by considerably fewer (but bigger) transactions than *genome*.
- ***labyrinth***: this is a variant of Lee's routing algorithm [13]. It has very different characteristics: short transactions (few instructions inside transactions) but it is composed by an extremely large number of transactions in comparison to the others.

We have executed the selected STAMP applications with three state-of-art STM libraries, namely TinySTM [5], TL2 [3] and SwissTM [4]. We have used the same SMP machine described in the previous section, running all experiments with 2, 4, 8 and 16 threads (in STAMP, the number of threads must be 2^n). All results were also obtained through the average of 30 executions (an insignificant standard deviation has also been observed).

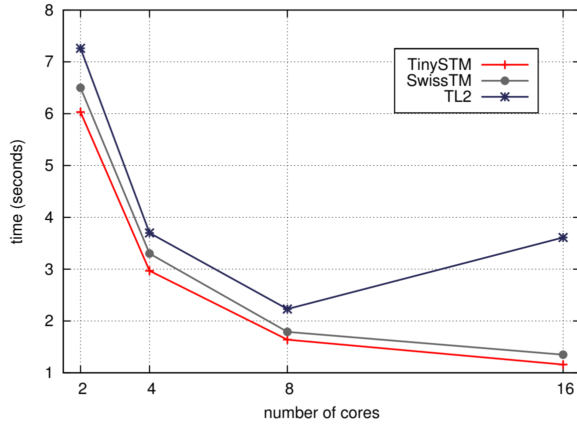

Figure 3. Execution times - Genome.

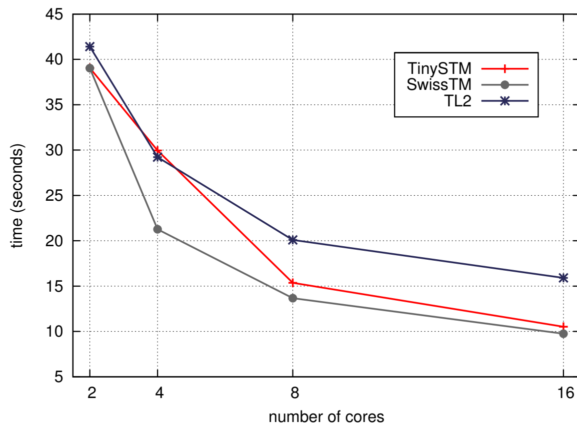

Figure 4. Execution times - Labyrinth.

Figure 3 shows the execution times we have obtained by executing *genome* with the three STM libraries. For this particular application, TinySTM has always presented lower

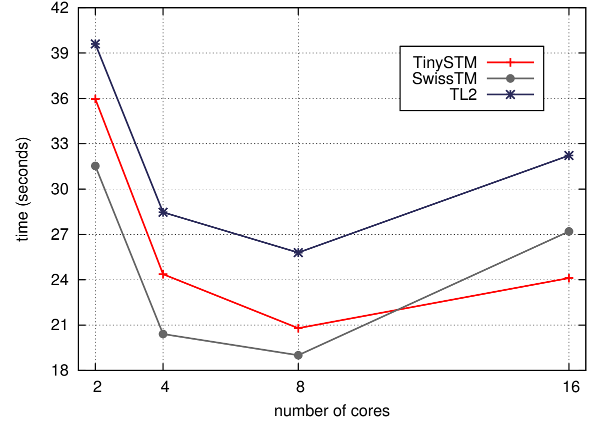

Figure 5. Execution times - Intruder.

execution times ($1.1s$ with 16 cores) and better scalability than the others. We can notice that TinySTM and SwissTM perform still better with 16 cores, while TL2, on the contrary, presents a considerable performance degradation.

The results of the *labyrinth* executions are shown in Figure 4. Unlike the previous results, SwissTM has presented better results than the others ($9.75s$ with 16 cores). TL2 has shown better scalability in comparison to the *genome* results but it has still presented poor performance compared to the other STM libraries.

Finally, in Figure 5, we show the execution times obtained while executing *intruder*. We can observe a very different behavior: all three STM libraries fail to scale with 16 cores.

Regarding the results presented above we can conclude that it is not trivial to predict the performance of a TM application. The characteristics and design choices of the STM library can undoubtedly change its performance. In this context, we argue that such characteristics must be taken into account in order to develop a performant TM application. Thus, it emerges the necessity of having ways to better comprehend their behavior. In order to obtain some insight on such issues, we propose an approach for collecting and tracing relevant information about transactions.

V. TRACING TRANSACTIONS

Tracing applications basically consists in recording a chronological history of events, representing the application behavior. An event is an action during the execution of an application that changes its state. In this work, we are specifically interested in events deriving from the use of STM. In the following sections, we describe how these events can be traced. Firstly, we specify the desired characteristics of our approach. Secondly, we explain the general functioning of STM libraries and what events we intend to trace. Finally, we describe our tracing mechanism.

A. Goals

Our mechanism aims to tackle two relevant issues:

- ***Application and STM library independency***: we want a tracing solution which does not change neither the TM application nor the STM library source codes. In order to do so, we have chosen to implement an interception mechanism placed between the STM application and the STM library.
- ***Low Intrusiveness***: our tracing solution should minimize intrusiveness meaning that it should not imply an important execution overhead. Indeed, when that overhead is important, the application behaves differently and the traces may not represent the real application behavior. In order to minimize intrusiveness, we have decided trace a very reduced set of events.

B. Traced Events

We have selected the two most important functions to be traced, *i.e.*, `stm_start()` and `stm_commit()`, which respectively indicate the beginning and the end of transactions. We believe that, by intercepting them, relevant information about TM applications can be extracted without increasing the degree of intrusiveness.

The function `stm_start()` is responsible for initializing the transaction specific structures, as well as for saving the calling environment for later use in case of abort. It typically saves the stack context containing the current thread local program counter and registers values. The role of the `stm_commit()` function is to verify whether the current transaction is in conflict with any other transaction.

When a conflict occurs, a rollback mechanism calls `stm_start()` in order to restart the transaction. The system rollbacks the transaction by restoring the environment that has been saved by the first call to `stm_start()`. In order to do that, `stm_start()` and `stm_commit()` use the system calls `sigsetjmp()` and `siglongjmp()` to respectively save and restore the thread environment. This strategy is applied in the majority of STM libraries, including TinySTM, TL2 and SwissTM. When there is no conflicts, all changes that have been done by the transaction are made permanent (*validation*).

C. Tracing Mechanism

Our tracing solution relies on the Linux dynamic linking mechanism which provides a simple way to intercept function calls. It provides the environment variable called `LD_PRELOAD` which is used to dynamically load a library LIB when launching applications. During the execution, the system will intercept the functions having the same signatures as the ones implemented in LIB, calling the corresponding LIB functions (*wrappers*). Wrapper functions may implement their proper behavior but it is still possible to call the original functions.

In this work, a shared library called `libTraceSTM.so` has been implemented, containing two wrappers for the `stm_start()` and `stm_commit()` functions. By executing `LD_PRELOAD=./libTraceSTM.so app` (where `app` is the target STM application), the original STM functions are dynamically overridden by our wrapper functions. The wrappers are responsible for tracing and calling the corresponding original functions, *i.e.*, `stm_start()` and `stm_commit()`.

Listing 3 shows the two selected wrapper functions: `stm_start()` (line 1) and `stm_commit()` (line 10). In `stm_start()`, we first obtain a handle to the original STM function (line 2), trace the related event using the `trace()` function (line 5) and call the original STM function (line 6). In `stm_commit()`, however, we first call the original STM function before tracing it. As explained before, `stm_commit()` performs a rollback in case of conflicts (`stm_start()` is called by the rollback mechanism). Thus, by tracing events after `stm_commit()` we ensure that transactions have already committed successfully.

```
1  void stm_start() {
2     realStmStart = dlsym(handle, "stm_start");
3     ...
4     pthread_mutex_lock(&trace_lock);
5     trace("stm_start");
6     (*realStmStart)(); //calls the real function
7     pthread_mutex_unlock(&trace_lock);
8  }
9
10 void stm_commit() {
11    realStmCommit = dlsym(handle, "stm_commit");
12
13    pthread_mutex_lock(&trace_lock);
14    ...
15    (*realStmCommit)(); //calls the real function
16    ...
17    trace("stm_commit");
18    pthread_mutex_unlock(&trace_lock);
19 }
```

Listing 3. STM Function Wrappers.

It is important to notice that calls to `trace()` and the real STM functions must be **atomic**. Otherwise, the order of the recorded events may not correspond to the real sequence of calls to STM functions. This is the reason why we use locks and some additional treatments in order to avoid deadlock situations that can easily arise during rollbacks.

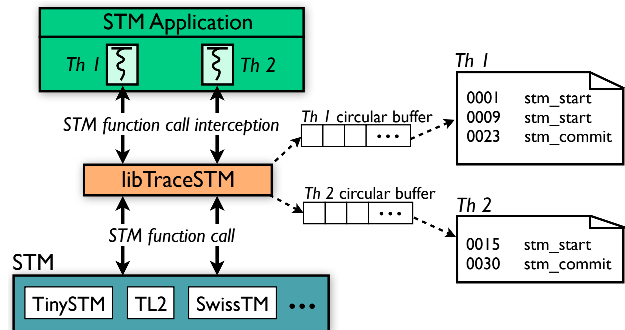

Figure 6. Overview of the Tracing Mechanism.

Figure 6 shows our tracing mechanism. When a thread is initialized by the STM application, our shared library

adds the thread ID in an internal data structure, creates a trace file and instantiates a circular memory buffer. When subsequent STM function calls are intercepted, the trace record is written into the corresponding thread's circular buffer. When the circular buffer is full, its contents are flushed to the corresponding trace file.

Each event to be traced is represented by a timestamp and the name of the intercepted function. As we target shared memory multithreaded applications, timestamps correspond to the value of the machine's clock. At the end of the execution, we obtain a set of files, one per thread. This is illustrated in Figure 6, which shows a STM application with 2 threads (named $Th1$ and $Th2$). The trace file of $Th1$, for example, shows two successive calls to `stm_start()` followed by a `stm_commit()`, which indicates that the transaction has been aborted once. The trace file of $Th2$ shows a transaction that has been started and committed successfully.

After the execution of the application, we perform a merge sort of the individual trace files considering their timestamps. In the merged trace file, each event is represented by a timestamp, a thread ID and the name of the intercepted function. Figure 7 shows the result of the merge sort considering two individual trace files from threads $Th1$ and $Th2$.

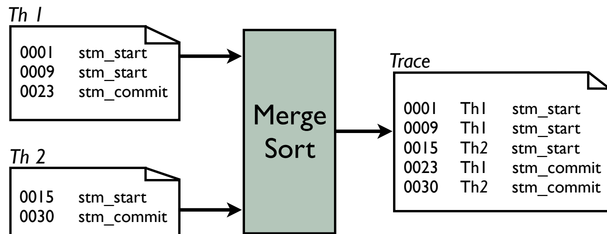

Figure 7. Merge Sort of Individual Trace Files.

We believe that our approach to intercept STM function calls is very simple and it can be easily extended if more functions should be traced. It is also generic enough, since it can be used with all STM libraries (*e.g.*, TinySTM, TL2 and SwissTM) and it does not change neither the STM application nor the STM library source codes.

VI. Experimental Results

This section presents the experimental results we have obtained by using our tracing mechanism with the STAMP applications. We have carried out experiments of all STM applications available from STAMP, but in this paper we present only the results we have obtained from the applications described in Section IV. We have decided to use TinySTM as the target STM library for all experiments presented in this section, executing the three applications with 16 threads.

Our tracing mechanism allows us to obtain different metrics and statistics about the execution of STM applications. For instance, we can calculate the number of transactions or the number of commits and aborts. We can also observe the wasted work, *i.e.*, the percentage of the transactions execution time that has been spent executing aborted transactions (total and per thread). Other accessible metrics concern the evolution of the number of aborts and commits and the instantaneous commit rate (the proportion of committed transactions at sample points) during the execution. In this paper, we present the evolution of the number of commits and aborts during the execution. We believe that such information can be very helpful since the number of aborts is one of the most important metrics that influences the performance of STM applications.

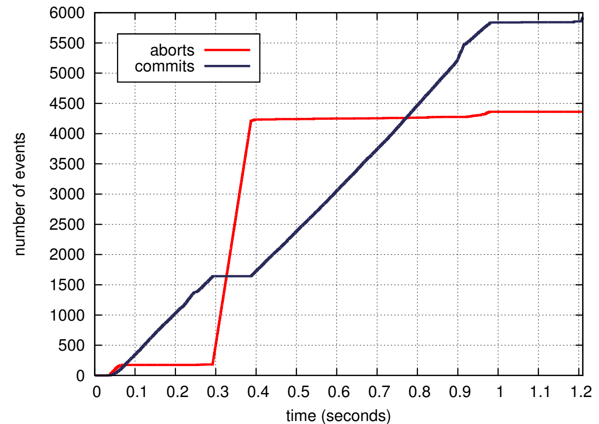

Figure 8. Commits and Aborts - Genome.

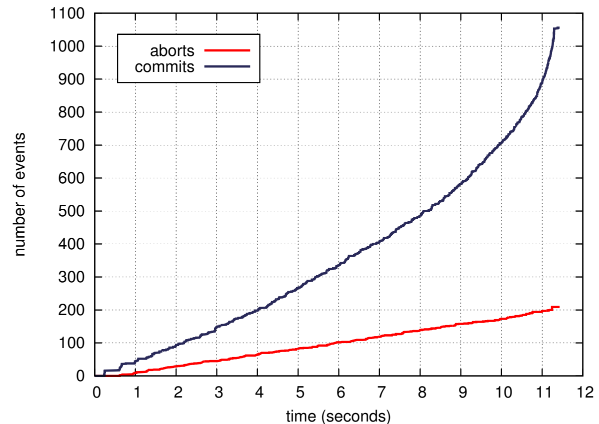

Figure 9. Commits and Aborts - Labyrinth.

Figure 8 concerns the *genome* application. What may be observed is that the commit/abort behavior changes during the execution. Namely, the number of aborts increases drastically between $0.3s$ and $0.4s$. This suggests that, during this period, the probability of having conflicting transactions is very high. However, since the time period is short compared to the total execution time, it does not

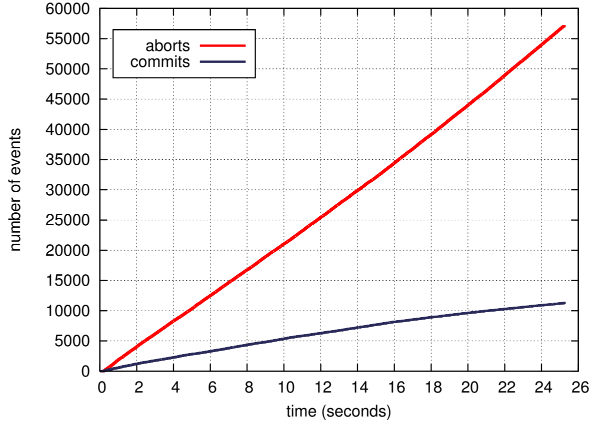

Figure 10. Commits and Aborts - Intruder.

interfere considerably with its overall performance (as seen in Section IV, Figure 3).

The results obtained with *labyrinth* are shown in Figure 9. As it can be seen, this application takes advantage of the optimistic approach of TM, since the aborts curve is always placed under the commits curve. The difference increases towards the end of the execution (exponential growth). The few number of aborts in comparison to the commits justifies the performance obtained in Section IV.

Finally, the results for *intruder* are shown in Figure 10. Its poor performance observed in Section IV is confirmed by the presence of a very high number of aborts in comparison to commits, which means that it does not take advantage of the TM optimistic approach.

We have also measured the intrusiveness of our tracing mechanism considering two important metrics: the *execution time* and the *number of aborts*. If execution times and number of aborts obtained by executing the STM applications with the tracing mechanism are very distinct in comparison to the original ones, we can conclude that our trace mechanism is very intrusive and it may change significantly the behavior of the applications. Table I shows such information, comparing these two metrics after executing all applications with and without the tracing mechanism.

Table I
INTRUSIVENESS OF THE TRACING MECHANISM.

	Genome		**Labyrinth**		**Intruder**	
	normal	*trace*	*normal*	*trace*	*normal*	*trace*
Time (s)	1.10	1.21	10.37	11.51	24.03	25.70
Intrusiveness	**9.09%**		**9.90%**		**6.5%**	
Aborts	4,362	4,030	209	205	57,058	54,805
Intrusiveness	**7.63%**		**1.91%**		**3.95%**	

As it can be observed, our trace mechanism was not considerably intrusive when comparing both metrics (execution times and number of aborts). For all experiments, we have obtained a maximum intrusiveness of 9.90% and 7.63% considering the execution time and the number of aborts, respectively. This means that the collected information about transactions by our tracing mechanism represent the execution behavior of the analyzed STM applications.

VII. Related Work

Considering that TM is an emerging research area, in the majority of cases, works have been based on proposals of different TM solutions and algorithms. Recently, some works have addressed the performance analysis of different TM solutions and/or TM applications, as in [12] and [14]. However, few works have been done concerning tools to help the development using TM. That is the case of the proposals presented in [15] and [16].

Minh *et al.* [12] have described the eight non-trivial STAMP applications, showing their performance gains with different TM systems and configurations. However, they have used a multicore simulator for all experiments instead of a real multicore platform. Marathe *et al.* [17], on the other hand, have compared the performance of their STM solution with other STM implementations on two multicore machines. The performance analysis has been based on four simple micro-benchmarks, which do not represent the behavior of real applications. Chung *et al.* [14] have studied 35 benchmarks from different domains. In that work, the authors have translated the original synchronization mechanisms applied on all benchmarks to TM. However, they have neither studied the non-trivial TM applications presented here nor they have analyzed aborts and commits, which are very important metrics.

Our work differs from these three, since we have tackled both issues: a performance analysis of non-trivial TM applications by using three different state-of-art STM libraries over a real multicore machine.

Ansari *et al.* [15] have manually instrumented the DSTM2 STM library to collect relevant information during the execution of the applications. They have chosen three applications from STAMP and an implementation of Lee's routing algorithm, investigating some relevant metrics to comprehend TM applications. Lourenço *et al.* [16] have proposed a monitoring framework, which collects the transactional events into a log file as well as a tool to visualize the results. Their instrumentation mechanism is based on a API, so the user must insert the tracing function calls within applications source codes.

Unlike these two proposals, our solution uses an interception approach based on the Linux dynamic linking mechanism. By using such method, we can achieve the STM application and STM library independency, since it does not change neither the TM application nor the STM library source codes and it can be easily applied with different STM libraries.

VIII. Conclusion and Perspectives

In this paper, we have shown that the performance of applications based on STM is related to two issues: the application itself and the STM library. A TM application that takes into account the characteristics of the underlying TM system may benefit of its optimistic approach, reducing the probability of having conflicts and then, resulting in better performance. On the other hand, we have seen that an application may also behave differently depending on the STM library, which means that the developer must be aware of how the STM library works to achieve the desirable performance.

In order to obtain a better understanding of the performance of STM applications, we have proposed an approach for collecting relevant information about transactions. It is based on a shared library which is dynamically linked with the STM application. Events to be traced are implemented as wrapper functions, which can be easily extended if other events must be traced. Moreover, our solution can be applied to different STM libraries and applications as it does not modify neither the target application nor the STM library source codes.

The collected information representing each event allows a general comprehension about the behavior of all transactions. However, we cannot correlate each event to its corresponding transaction in the context of a specific thread, since transactions are not explicitly identified. As a future work, we intend to study ways of discerning transactions in our tracing mechanism, so finer information can be obtained. Moreover, we aim at investigating the behavior of other non-trivial TM applications, proposing general guidelines to reduce conflicts by analyzing TM applications. Finally, we also plan to study what support we would need in order to use our trace mechanism with hardware and hybrid TM solutions.

References

[1] J. Larus and C. Kozyrakis, "Transactional Memory: Is TM the Answer for Improving Parallel Programming?" *Communications of ACM*, vol. 51, no. 7, pp. 80–88, 2008.

[2] J. Larus and R. Rajwar, *Transactional Memory (Synthesis Lectures on Computer Architecture)*, 1st ed. Madison, USA: Morgan & Claypool Publishers, 2007.

[3] O. S. D. Dice and N. Shavit, "Transactional Locking II," in *DISC '06: Proc. of the 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[4] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching Transactional Memory," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 155–165, 2009.

[5] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 237–246.

[6] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun, "Characterization of TCC on Chip-Multiprocessors," in *PACT '05: Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[7] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based Transactional Memory," in *HPCA '06: Proc. of the 12th International Symposium on High-Performance Computer Architecture*, 2006, pp. 254–265.

[8] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid Transactional Memory," in *PPoPP '06: Proc. of Symposium on Principles and Practice of Parallel Programming*, 2006.

[9] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. S. III, and M. F. Spear, "Hardware Acceleration of Software Transactional Memory," in *TRANSACT '06: Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compiler and Hardware Support for Transactional Computing*, 2006.

[10] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why Is It Only a Research Toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[11] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press, 2007.

[12] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC '08: Proc. of The IEEE International Symposium on Workload Characterization*, 2008.

[13] X. Ji-Guang and T. Kozawa, "An Algorithm for Searching Shortest Path by Propagating Wave Fronts in Four Quadrants," in *DAC '81: Proc. of the 18th Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 1981, pp. 29–36.

[14] J. Chung, H. Chafi, C. C. Minh, A. Mcdonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "The Common Case Transactional Behavior of Multithreaded Programs," in *HPCA '06: Proc. of the 12th International Conference on High-Performance Computer Architecture*. IEEE Computer Society, 2006.

[15] M. Ansari, K. Jarvis, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson, "Profiling Transactional Memory Applications," in *PDP '09: Proc. of the 17th International Conference on Parallel, Distributed, and Network-based Processing*, 2009, pp. 11–20.

[16] J. Lourenço, R. Dias, and J. Luís, "Understanding the Behavior of Transactional Memory Applications," in *PADTAD '09: Proc. of the 2009 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2009.

[17] V. J. Marathe and M. Moir, "Toward High Performance Nonblocking Software Transactional Memory," in *PPoPP '08: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2008, pp. 227–236.