Software Transactional Memory: Worst Case Execution Time Analysis

Toufik Sarni and Audrey Queudet
LINA - University of Nantes
France
FirstName.LastName@univ-nantes.fr

Patrick Valduriez
LINA and INRIA
Nantes - France
Patrick.Valduriez@inria.fr

Abstract

While real-time applications are becoming more and more concurrent and complex, the drive toward multicore systems raises new challenges related to the parallelization of such performance-critical applications. Transactional memory is an attractive concept for expressing parallelism for programming multicore systems as it avoids the problems of lock-based methods and eases programming. However, it has not yet been exploited for real-time systems. In this paper, we propose the first real-time directed case study of software transactional memory. In particular, our goal is to identify the origin of the variation of the worst-case execution times (WCET) of transactions in memory. Based on a real implementation, we show through various experiments that for soft real-time, transactions rollback times are not the main cause of execution times variation. A good memory allocator must also be provided in order to suitably bound the WCETs of transactions into software transactional memory.

1 Introduction

With the advent of multicore systems, the transactional memory (TM) concept has attracted much interest from both academy [1, 2] and industry [3] as it eases programming and avoids the problems of lock-based methods. By supporting the ACI (Atomicity, Consistency and Isolation) properties of transactions, TM relieves the programmer from dealing with locks to access resources. More important, it avoids the severe problems of lock-based methods such as deadlock situations and priority inversions. While lock-based methods systematically block all accesses to shared resources, transactional memory allows several transactions to access resources in parallel. A transaction is either aborted when a conflict is detected, or committed in case of successful completion. Conflicts are handled with non-blocking synchronization which offers a stronger guarantee of forward progress.

There are three kinds of implementations for transactional memory: hardware-based memory (HTM) [1, 4], software-based memory, denoted as software transactional memories (STM) [2, 5, 6, 7] and hybrid schemes (HyTM) that combine both hardware and software supports [8, 9]. HTM researchers mainly focus on implementation with less attention to performance. On the contrary, STM researchers take care about performance issues on TM, and several policies [10, 11] have been proposed to manage conflicts between transactions.

While software transactional memories are widely studied for numerous and various purposes, they have not yet been studied for real-time systems. However, we believe that the advantages of transactional memory can also be brought to real-time systems. Thus, we propose to study how to adapt it to soft real-time systems. For this purpose, we aim to identify which parts of STM cause WCET variations. It is often claimed that transaction rollback times are the main cause of unpredictability in execution times. However, the recent STMs are usually dynamic memory based. We show in this paper that STM memory allocators require more consideration than rollback times in order to bound the execution time of transactions. Furthermore, we show that transaction rollback times also depend on the time latencies of the underlying operating system. This is why we focused on selecting the best task scheduling policy minimizing the rollback times.

To the best of our knowledge, this paper is the first to study the WCET variation of STM based on a real implementation. The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the real-time scheduling of both tasks and transactions and presents the STM used in our experiments. Section IV presents both the issues identified for adapting STM to satisfy real-time constraints and their implementation. Section V gives an experimental analysis under several real-time scheduling policies of tasks and shows the impact of memory allocator on the STM. Finally, Section VI draws the main conclusions and discusses future work.

2 Related Work

Schoeberl *et al.* [12] propose a real-time HTM. This HTM use the late conflict detection (*i.e* the conflict between transactions is detected on a commit). The transaction is either rollbacked on a conflict or aborted on context switch. The number of retries of the transaction is bounded and integrated into the WCET analysis. This bound assumes one atomic region per thread period and allows having hard real-time constraints. However, we are interested by soft real-time STM, and the HTM presented by the authors assume that all critical sections resources need to be known.

Brandenburg *et al.* [13] compare wait-free and lock-free algorithms with spin-based and suspension-based synchronization mechanisms. They conduct experiments[1] using the real-time operating system LITMUS^{RT}. The four approaches are compared on the basis of both schedulability and tardiness bounds, by evaluating their respective overheads with respect to job release, scheduling and context-switching. One of the major conclusions of this work is that non-blocking algorithms are generally preferable for small, simple shared objects. Among non-blocking approaches, the authors conclude that wait-free algorithms are preferable to lock-free algorithms. Regarding scheduling policies, they show that, unlike P-EDF, the G-EDF policy does not scale for lock-free algorithms when the access to shared objects occurs at high frequency.
The wait-free algorithms are primarily of interest in hard real-time transactions [14]. However, implementing a wait-free-based STM is very difficult since fair access to memory is usually not guaranteed.

Riegel *et al.* [15] deal with time-based transactional memory that uses time to reason about the consistency of the data accessed by transactions and the order in which transactions commit. Usually, implementations like [16, 17] rely upon shared counters which can quickly become bottlenecks as the number of concurrent threads grows.
Riegel *et al.* [15] show how a time base can affect transactional memory performance. They rely on experiments[2] which compare the use of a shared integer counter with that of a MMTimer which is a real-time clock with an interface similar to the High Precision Event Timer widely available in x86 machines. Their main observation is that this enhanced hardware support can ensure a much better clock synchronization than mechanisms that require communication via shared memory. As part of their work, the authors introduce the *Real-Time Lazy Snapshot Algorithm* (LSA-RT) which is a timestamp-based algorithm using a real-time clock. Moreover it uses a helper mechanism to help committing transactions to complete. However, the authors consider only throughput, and not WCET of transactions. Furthermore, they consider the time-based STM performance without tacking into consideration the impact of the operating system in which their STM is performed.

Yoo *et al.* [18] describe a scheduler for transactional memory. The authors compare their *adaptive transaction scheduler* to the traditional *Contention Manager* (CM). In CM-based STMs [19, 11], the transaction that encounters a conflict, consults its CM. When the CM retries the denied object, it typically employs an exponentially backoff scheme with a retry interval expanding exponentially to a maximum limit until success. Thus, a CM can decide to abort a certain transaction, but does not deal with *when* to resume an aborted transaction. In contrast, the scheduler presented by the authors, specially deals with *when* to resume the aborted transaction which is an important notion in a real-time context. However, the authors do not deal with any real-time constraints in their paper.

3 Theoretical Background

3.1 Real-Time Task Model

We consider the scheduling of a sporadic task system τ on $m \geq 1$ processors. For each task $\tau_i \in \tau$ we associate a set of jobs $J = \{j_1, j_2, ..., j_n\}$. Task τ_i is characterized by a set of parameters r_i, C_i, P_i which respectively represent the task release, its execution requirement in the worst-case, and its period of activation. At time $r_i + (k-1)P_i$ and for $k \geq 1$, a k^{th} job is released, receives C_i units of processor time and should complete by its absolute deadline $D_i \geq r_i + kP_i$. The weight (or processor utilization) for a task τ_i on processor m is defined by $u_{i,m} = C_i/P_i$. We assume that at any time, a processor executes at most one job, and a job is executed at most on one processor.

Scheduling of tasks. On multiprocessor systems, two alternative paradigms for scheduling collections of tasks are considered: *partitioned* and *global* scheduling. In the partitioned approach, the tasks are statically assigned to processors and are always executed on a single processor. Each processor has its own scheduling queue of tasks which is independent of other processors and the migration of jobs or tasks on other processors is not allowed. Feasibility analysis under the partitioned paradigm which is comparable to a *bin-packing* problem, is NP-Hard. Indeed it consists in placing k objects with different sizes in m boxes which respectively represent the tasks and the processors in our case. First-Fit and Best-Fit algorithms and their variants [20] are usually used to assign tasks to processors with an appropriate condition in accordance with the schedulability analysis. In contrast, under the global scheduling approach, inter-processor migrations are allowed. A single queue and only one policy are applied to tasks. A known result for uniprocessors is that the scheduling algorithm Earliest Deadline First (EDF) is optimal [21]. Unfortunately, EDF is *not* optimal on multiprocessors either under the partitioned or the global approaches [22], called respectively P-EDF and G-EDF. Another class of scheduling algorithms, which differs from the previous ones, gathers the *Pfair* algorithms (namely PD and PD^2) [23]. These are based on the idea of *proportionate fairness* and ensure that each task is executed with uniform rate.

[1]The hardware platform used was a four 32-bit Intel Xeon(TM) processors running at 2.7 GHz

[2]using a 16-processor partition of an SGI Altix 3700 and a ccNUMA machine with Itanium II processors

Tasks are broken into quantum-length subtasks and time is subdivided into a sequence of subintervals of equal lengths called *windows*. A subtask must execute within the associated window and migration is allowed for each subtask. With respect to feasibility, the authors in [23] proved that a periodic task set with $r_i = 0$ has a Pfair schedule on m processors iff:

$$\sum_{\tau_i \in \tau} \frac{C_i}{P_i} \leq m \qquad (1)$$

In order to make our experimental evaluation, as complete as possible, we select one algorithm in each class of scheduling (*i.e.* P-EDF, G-EDF and PD^2). Although the PD^2 algorithm is used to schedule hard real-time tasks on multiprocessors, we choose to include it in our study so as to cover all kinds of real-time applications.

3.2 Real-Time Transactions

Like real-time tasks, real-time transactions are classified according to the criticity of their deadlines: *hard*, *soft* or *firm*. The hard[3] class is rarely considered. Most studies assume the scheduling of transactions either in soft[4] or firm[5]classes.

Scheduling of transactions. The scheduler of transactions in database systems embeds a *concurrency control* protocol, which is in charge of resolving the conflicts between transactions when they occur, in order to maintain database consistency. In real-time database systems, not only database consistency should be satisfied, but transactions must also meet their deadlines [24]. To our knowledge, no real-time concurrency control policies are specially designed for software transactional memories.

3.3 Fraser's STM

FSTM [25] is a dynamic lock-free object based STM. It has been implemented as a C library. FSTM employs a recursive helping and an enforced global total order for transactions to ensure that despite contention, at least one process is making progress. The object is the basic unit of concurrency. Each object is pointed by an *object header* which contains the current version of the object (see Fig. 1.). The object header is pointed by an object handle which keeps the old and new references to the object. In case of a successful commit, the object header is updated with the new data block object. The transaction descriptor embodies both read-only and read-write lists. When a transaction accesses an object, the procedure is similar for both read-only and read-write accesses. The data structures described above are thus created according to the type of access. A *shadow copy* of the object is also created in the case of a read-write access and remains private until the transaction commits.

[3]System cannot tolerate the missing of deadlines.

[4]The transaction could be accepted even if it misses its deadline.

[5]Missing the deadline causes to abort the transaction.

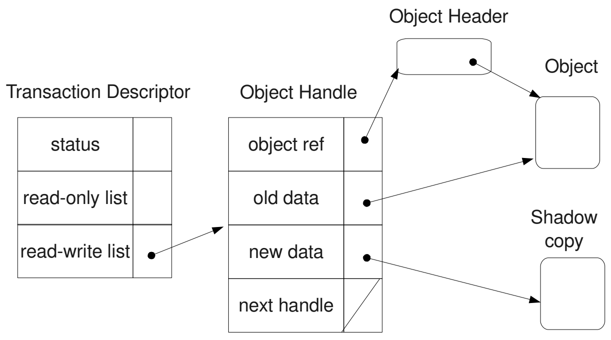

Fig. 1. Fraser's STM data structures

The commit phase is divided into three phases. The first phase is the acquire phase. The transaction attempts to acquire ownership of all objects on its read-write list in a canonical order. The transaction that attempts to acquire ownership of the object, performs a CAS (Compare And Swap) operation on the object header, to replace the pointer to the object by a pointer to its transaction descriptor. If the content of the object header points to a more recent object, the transaction will then abort. However, if the object is owned by another transaction then the obstruction is helped to completion. The second phase is the read phase. It checks whether each read-only object has not been updated since it was opened. If all objects are successfully acquired or checked then the transaction attempt to commit successfully. In the last phase, all acquired objects are released and if the transaction commits then all old objects are replaced by their corresponding shadow copy (*i.e.* the new object).

4 Introducing Real-Time into STM

We aim to implement a real-time STM with soft constraints by minimizing the execution time jitter of transactions. In order to make STM suitable for soft real-time, not only the rollback times should be taken into consideration, but also both the scheduler of transactions and that of the operating system. Therefore, we propose to analyze which parts of STM cause execution time variations. Static memory approaches as proposed in the first implementation of STM [2] could be a good candidate to bound the execution time of transactions, but only basic real-time applications are involved in this case. It therefore contradicts the transactional memory concept which is rather intented for complex applications. In our study, we are interested in taking into consideration the dynamic allocation of memory since most of the recent STM implementations integrate a garbage collector. However, the dynamic allocation of memory in real-time context, is usually avoided because considered as an unbounded part. To summarize, we have to face orthog-

onal constraints while considering complex real-time applications using dynamic memory-based STMs.
As a solution, we choose to enhance Fraser's STM because its scheduler is based on the recursive helping between transactions. The helping mechanism appears more suitable for soft real-time. Indeed, a transaction with a low priority can help a transaction with higher priority and then at least one transaction will make progress. Moreover, FSTM dynamically creates and deletes objects in memory. Other implementations of STM like DSTM [19] are not considered here. Indeed, DSTM is an obstruction-free based implementation which provides the weakest guarantee to make progress. Consequently, it is not suitable for real-time systems.

4.1 Implementation

Intuitively, the underlying operating system (OS) has to be considered since transactions are executed within threads. That is why we use a real-time operating system (RTOS) named LITMUS^{RT} [6] [26]. Designed to run on top of a symmetric multiprocessor (SMP) architecture, it implements all the real-time task scheduling algorithms described in Section 3.1. LITMUS^{RT} is based on the Linux operating system (kernel version 2.6.24). The proposed schedulers are implemented as plugin components that can be selected from Linux user-space. In order to manipulate both tasks and synchronization mechanisms from Linux user-space, system calls are gathered within a C library. For all these reasons, LITMUS^{RT} becomes an excellent (perhaps the only) candidate to study the behavior of FSTM on multiprocessor systems, under a panel of advanced real-time scheduling policies.

We use the TLSF (Two-Level Segregate Fit)[7] [27] memory allocator to show the impact of object's allocation within our WCET analysis. TLSF is based on an algorithm that has a constant cost $\Theta(1)$. It solves then the problem of the worst case bound, thus maintaining the efficiency of the allocation and deallocation operations. Therefore, TLSF allows the reasonable use of dynamic memory in real-time applications.

4.1.1 Integration into LITMUS^{RT} library

Under LITMUS^{RT}, a real-time task is initially created as a standard linux thread (using the standard *pthread* library) before being effectively started. Then, it initializes the real-time environment and specifies the real-time parameters of the task, namely C_i and P_i. Thereby, the thread sporadically releases its jobs by calling the job function every P_i units of time.
To summarize, FSTM and the LITMUS^{RT} library have been combined by creating real-time threads within FSTM. We performed this integration so as to support both non real-time threads and real-time tasks.

4.1.2 Integration of TLSF library

TLSF is a C library. We integrated it into FSTM by replacing all the allocation and deallocation functions by those provided by TLSF. The memory pool which is used by TLSF is created at initialization time by the classical *malloc* function. Note that the TLSF's initialization is done before the creation of real-time threads.

5 Experimental Evaluation

5.1 Evaluation Context

We present here the experiments we performed to evaluate FSTM with respect to WCETs. Firstly, we describe the hardware and software configurations we use for our experimental evaluation, as well as the STM benchmarks we consider. Secondly, we report comparative results allowing us to select the best scheduling policy among Linux and LITMUS^{RT} operating systems. Finally, we study the dynamic memory allocator impact on FSTM.

Hardware context. The hardware platform used in our experiments is a two 32-bit multicore Intel Core(TM)2 Duo T7500 processors running at 2.20GHz with 4MB L2 cache, and 3.5GB of main memory. During all experiments, the multicore option has been enabled, and the cpu frequency for each core has been fixed at 2194MHz.

Software context. We have compiled the LITMUS^{RT} kernel for the above hardware platform and used it on top of an Ubuntu 8.04 hardy Linux distribution. The system has never been overloaded during the experiments neither under Linux (*i.e* only the test application has been launched), nor under LITMUS^{RT}.

Real-time task parameters. For each real-time task, we fixed $C_i = 20ms$ and $m = 2$; the parameter P_i being determined according to Equation 1. Thus, in all cases, we consider processors under heavy loads. The impact of the variation of these parameters is not considered in this paper, and we defer its consideration for future work.

STM benchmark. The experiments performed by Fraser [25] for the performance evaluation of STMs are about 10 seconds of duration. Fraser considers that this duration is pretty sufficient to stabilize the data into the cache, since after 10 seconds the same values are repeated. During the 10s of test, the evaluated STM performs a series of three operations: readings, writings and deletes over the shared objects organized as red-black trees or skip lists. The proportion of each operation performed is given as an input parameter of the benchmark. Fraser thinks that 75% of reads and 25% of writes and deletes well reflect a real situation.
For our experiments we used only red-black trees. Each experimental test lasts 10 seconds and operations are composed of 75% of reads namely lookup and 25% of writes

[6] http://www.cs.unc.edu/∼anderson/litmus-rt
[7] http://rtportal.upv.es/rtmalloc/

and deletes namely update and delete respectively. Shared resources are highly contended, with 2^4 maximum keys for red-black trees. Note that we have slightly modified this benchmark in order both to adapt it to the real-time context and to make our measurements.
Unlike classical STMs in which performance evaluation usually uses the average number of transactions per success and per time duration, we use other parameters for our real-time evaluation. These are described below.

5.2 Performance Metrics

Transaction WCET jitter. We measure the execution time of the three operations usually performed by a transaction (i.e. lookup, update and remove). The transaction WCET for each operation corresponds to a mean value and is obtained over all launched threads for a test of 10 seconds duration. This test is repeated 10 times. The jitter is then the variation of the WCET observed between each test. To perform these measurements, we recover the current processor ticks by calling the assembly instruction *rdtsc*. Each operation time is obtained by subtracting the processor ticks value at the end of operation to that at its start time. However, this method to get the ticks value at user-level does not work. Indeed, if a transaction starts on one core and migrates to another core, then the execution of the transaction becomes invalid since the clockticks of the cores are not synchronized.
We have proposed an alternative solution (see Algorithm 1) which consists in adding the core identity to the context of the thread. This is done by calling the assembly instruction *cpuid*[8]. Secondly, we make sure that the CPUID is corresponding to the *rdtsc* (see line 6) as the instructions are not atomically executed.
If task migration occurs more than 2 times during the test then we stop the retries (line 7). According to the state in which we perform the test, either we abort the program at start time of transaction operation (line 9) or consider the test as a bad one at the end of operation (line 11). At the end of the experiment, if the number of transactions that have experimented bad test is up to 1% of the total number transactions, then the experiment is manually restarted.
Note that we have measured the time duration of Algorithm 1. which is 0.5μs. Thus, the worst case execution path of this algorithm is 2μs (*i.e*, 2×0.5 at the starting time of the transaction operation, plus 2×0.5 at the end of the operation). Therefore, the WCET has a precision within the interval $[1, 2]\mu$s.

Time variation factor. As the experiment that gives the WCET of transactions is repeated 10 times, we obtain 10 values of WCET for each one of the three operations of the transactions. For each operation value, we compute their mean $\overline{x}$ and their standard deviation σ. Let the time variation factor $V = \frac{\overline{x}}{\sigma}$. The variation facor V is then a ratio which provides information on the variation degree of the WCET of transactions over 10 experiments.

Rollback time ratio. This parameter is measured once

Algorithm 1 Transaction operation measurement
```
1:  $init\ RetryCPU \Leftarrow 2$
2:  $Thread_i \rightarrow coreID \Leftarrow CPUID()$
3:  repeat
4:      $RetryCPU \Leftarrow RetryCPU - 1$
5:      $Thread_i \rightarrow TrOperation_j \Leftarrow ReadTicks()$
6:  until $((Thread_i \rightarrow coreID = CPUID())$ and $(RetryCPU \neq 0))$
7:  if $RetryCPU = 0$ then
8:      if $state = StartTrOperation$ then
9:          $Abort()$
10:     else
11:         $BadTest \Leftarrow BadTest + 1$
12:     end if
13: end if
```

and the experiment does not repeat (*i.e* 10 seconds of duration only). We define for each thread, the rollback-time ratio $roll_i$ of its transactions. For each operation O_i of the transaction, the parameter $roll_i$ is defined as follows: $roll_i = \frac{\sum^n RollbackTime(O_i)}{\sum^n Duration(O_i)}$. The global rollback-time R we consider for our experiments is then:

$$R = \frac{\sum^N roll_i}{N} \tag{2}$$

where N is the number of threads.

5.3 Results

5.3.1 OS's impact

In this experiment, we intent to show how the underlying operating system can impact on the rollback times of transactions. Results are presented in Figures 2, 3 and 4. Note that the average number of transactions is around of 7×10^6 for each case.

We can see that the parameter R is constant and less than 0.25% for the three policies, namely Linux, G-EDF, and PD^2. This value can be practically ignored since in each policy it still remains constant for an increasing number of threads.
We observe that for the Pfair policy (see Fig. 3), R is reduced. This is because Pfair does not scale due to its important migration cost. Indeed, the migration cost increases the effective duration time of the thread and thereby that of transactions. Transaction rollbacks rarely occur and then are less likely to be impacted by the migrations. In fact, the values used for computing R – not presented here for readability – show that the rollback time is not reduced, but only the duration of transactions is increased. The same phenomenon can be slightly observed with G-EDF (see Fig. 4) since the G-EDF ratio of migrations is usually lesser than that of Pfair.
On the contrary, Fig. 5 shows that R is almost null. In this case, the duration of transactions is relatively reduced thanks to the minimal overheads induced by P-EDF.

[8]The id assigned by the APIC is at the 25-bit in our case

These overheads, mainly caused by task migrations, are avoided under P-EDF, thus minimizing rollback times.
Therefore, this experiment shows that under FSTM, rollback times do not make up the major part of the transaction duration. In addition, according to their weak impact, rollback times can be ignored when doing the WCET analysis for soft real-time constraints. Furthermore, for the reasons mentioned above, we choose the P-EDF policy for the rest of our experiments.

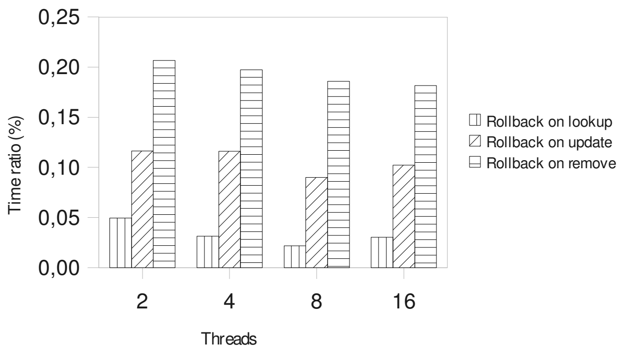

Fig. 2. Rollbacks under Linux

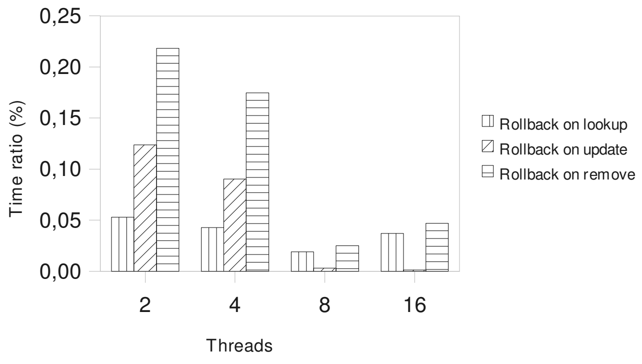

Fig. 3. Rollbacks under PD^2

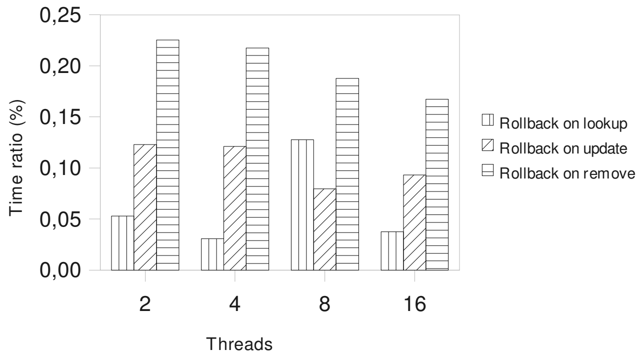

Fig. 4. Rollbacks under G-EDF

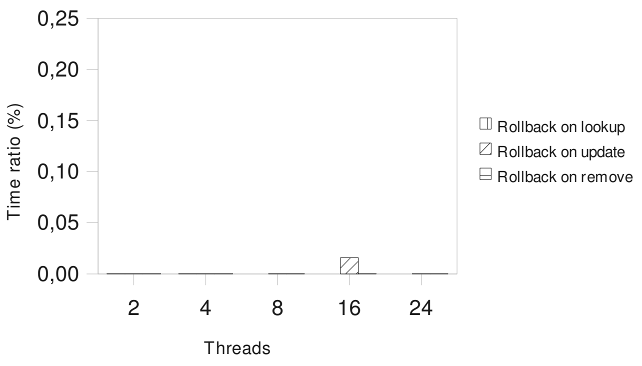

Fig. 5. Rollbacks under P-EDF

5.3.2 Dynamic memory allocator's impact

Since rollbacks do not impact significantly on the duration of transactions, we attempt here to show which part has really a detrimental effect, considering the P-EDF policy (selected from the previous experiment). We compare the results obtained using the classical memory allocator *malloc* with that of TLSF, on the basis of the V parameter defined above.

Fig. 6. shows that the duration of transactions has an important jitter for the three operations. Although P-EDF is used, FSTM suffers from important time latencies that characterize the execution environment at each program launch. FSTM uses a garbage collector that we have configured to be in minimal mode. Indeed, the normal mode often causes the program to abort due to a chunk imposed not only to deaden the cost allocation but also to increase the per-cacheline pointer density. However, we noticed that this mode of garbage collector configuration impacts on the total memory used by FSTM, but not on the V parameter.
The real reason of this variation is demonstrated on Fig. 7 and Fig. 8. When TLSF is used instead of the classical memory allocator *malloc*, the WCET of transactions is bounded with almost the same value. Indeed, the maximum variation that is reached using *malloc* is around 160% versus 8% when using TLSF.
This shows that FSTM could satisfy soft real-time constraints provided a bounded memory access is performed (i.e. using a constant-time dynamic memory allocator like TLSF).

6 Conclusion

We believe that the advantages of transactional memory can also be brought to real-time systems. Thus, we studied the possibility of introducing soft real-time into STMs by analyzing the WCET of transactions. To our knowledge, such study has not been attempted before.
The main results of our study are summarized hereafter: **(i)** P-EDF reduces the rollback times of transactions; **(ii)** For soft real-time constraints, the rollback times could be ignored within FSTM when doing the WCET analysis;

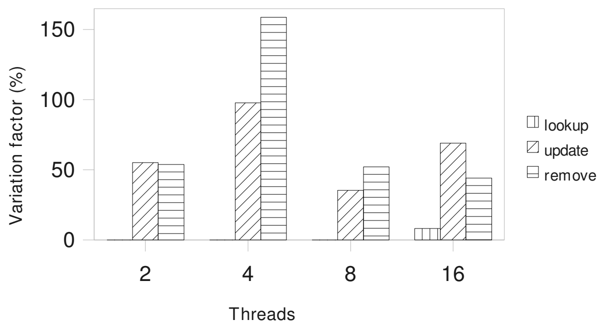

Fig. 6. WCET jitter using classical *malloc* (P-EDF)

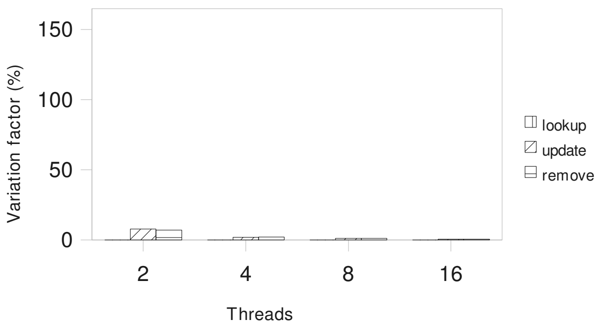

Fig. 7. WCET jitter using TLSF (P-EDF)

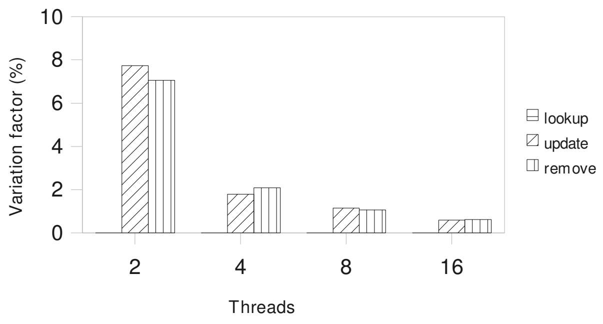

Fig. 8. Zoom on Fig. 7

(iii) FSTM could greatly satisfy soft real-time constraints provided memory accesses are bounded.
Now that we have bounded time in STM, many directions are then possible for future work. First, in this study we only dealt with the duration of transactions. It would be interesting to study the impact of STM on the number of deadline violations when scheduling real-time transactions. Secondly, in our experiments, we arbitrarily fixed the parameters of the real-time tasks. It would be also interesting to evaluate the impact of the processor load. Finally, we would like to formalize the interaction between the real-time scheduler of tasks and that of transactions.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *proc. the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[2] N. Shavit and D. Touitou, "Software transactional memory," in *proc. the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1995, pp. 204–213.

[3] M. Tremblay and S. Chaudhry, "A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc r processor," *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory." in *HPCA*. IEEE Computer Society, 2005, pp. 316–327.

[5] R. Ennals, "Softawre transactional memory should not be obstruction-free," Intel Research Cambridge, Tech. Rep., 2006.

[6] K. Fraser and T. Harris, "Concurrent programming without locks." *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007.

[7] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrt-stm: a high performance software transactional memory system for a multi-core runtime." in *PPOPP*, J. Torrellas and S. Chatterjee, Eds. ACM, 2006, pp. 187–197.

[8] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory." in *PPOPP*, J. Torrellas and S. Chatterjee, Eds. ACM, 2006, pp. 209–220.

[9] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory." in *ASPLOS*, J. P. Shen and M. Martonosi, Eds. ACM, 2006, pp. 336–346.

[10] W. N. Scherer III and M. L. Scott, "Contention management in dynamic software transactional memory," in *proc. the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, Jul 2004.

[11] W. N. S. III and M. L. Scott, "Advanced contention management for dynamic software transactional memory." in *PODC*, M. K. Aguilera and J. Aspnes, Eds. ACM, 2005, pp. 240–248.

[12] M. Schoeberl, B. Thomsen, and L. L. Tomsen, "Towards transactional memory for real-time systems,"

Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Research Report 19/2009, 2009.

[13] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2008, pp. 342–353.

[14] J. H. Anderson, R. Jain, and S. Ramamurthy, "Implementing hard real-time transactions on multiprocessors," in *RTDB*, 1997, pp. 247–260.

[15] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 221–228.

[16] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii." in *DISC*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 194–208.

[17] M. F. Spear, V. J. Marathe, W. N. S. III, and M. L. Scott, "Conflict detection and validation strategies for software transactional memory." in *DISC*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 179–193.

[18] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems." in *SPAA*, F. M. auf der Heide and N. Shavit, Eds. ACM, 2008, pp. 169–178.

[19] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III, "Software transactional memory for dynamic-sized data structures." in *PODC*, 2003, pp. 92–101.

[20] D. Johnson, "Fast algorithms for bin packing," *Journal of Computer ans Systems Science*, vol. 8, no. 3, pp. 272–314, 1974.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[22] M. L. Dertouzos and A. K. Mok, "Multiprocessor on-line scheduling of hard-real-time tasks." *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1497–1506, 1989.

[23] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[24] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: a performance evaluation," in *VLDB*, 1988, pp. 1–12.

[25] K. Fraser, "Practical lock freedom," Ph.D. dissertation, Cambridge University Computer Laboratory, 2003, also available as Technical Report UCAM-CL-TR-579.

[26] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmusˆrt : A testbed for empirically comparing real-time multiprocessor schedulers." in *RTSS*. IEEE Computer Society, 2006, pp. 111–126.

[27] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: A new dynamic memory allocator for real-time systems," in *ECRTS*, 2004, pp. 79–86.