Real-Time Support for Software Transactional Memory

Toufik Sarni*, Audrey Queudet*, Patrick Valduriez†
University of Nantes*
LINA and INRIA, Nantes, France*†
emails: {FirstName.LastName@univ-nantes.fr}* Patrick.Valduriez@inria.fr

***Abstract*—Transactional memory is currently a hot research topic, having attracted the focus of both academic researchers and development groups at companies. Indeed, the concept of transactional memory has recently attracted much interest for multicore systems as it eases programming and avoids the problems of lock-based methods. However, up to now, the scheduling of real-time transactions within software transactional memories has not been studied. To address this issue, we present in this paper a real-time software transactional memory, namely RT-STM. We focus on the scheduling of concurrent soft real-time transactions. In particular, we explore a new heuristic for conflict resolution that reduces the number of deadline violations when scheduling soft real-time transactions. After having discussed the scalability of various classical STMs under a real-time operating system, we present experimental results that show that RT-STM can improve the performance of transactional memory-based applications on multicore platforms.**

I. 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. 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 (HTM) [1], [4], software-based ones, 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 the 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.

There is an early attempt towards hard real-time for HTM [12]. However, up to now, no real-time transaction model has been specially defined for STM. While real-time scheduling of transactions has been widely studied in real-time databases, the issue has not yet been addressed for TM. Most of existing solutions for real-time scheduling consider either tasks in multiprocessor systems or transactions in database systems, but not both together. Therefore, we have turned our attention towards the design of a real-time STM in which we have formalized the introduction of a real-time model. We have implemented a new real-time scheduler of transactions for resolving conflicts between concurrent soft real-time transactions. The transaction model combines concepts found for the management of real-time transactions in databases and real-time tasks on multiprocessors. The main characteristic of this model is that it considers deadlines for transactions. This deadline is used by the scheduler either to abort or to help a transaction to complete.

To the best of our knowledge, this paper is the first to introduce real-time scheduling of transactions into STM. The rest of the paper is organized as follows. Section II discusses related work. Section III introduces the models of both tasks and transactions and presents the various STMs used in our experiments. Section IV presents our real-time transaction model and its implementation within transactional memory. Section V gives an experimental analysis of STMs under several real-time scheduling policies of tasks and shows the advantages of our solution. Finally, Section VI draws the main conclusions and discusses future works.

II. Related Work

Brandenburg *et al.* [13] compare wait-free and lock-free algorithms with spin-based and suspension-based synchronization mechanisms. They conducte 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 partitioned EDF, the global EDF policy does not scale for lock-free algorithms when the access to shared objects occurs at high frequency.

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

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 timestamp mechanism is not suitable for real-time transactions. Indeed, the timestamp represents the arrival time of a transaction but does not provide any information about its time constraint. In addition, the conflict resolution is performed according to the evaluation of the timestamp-based age of the transactions and yet, in a real-time context, a recent transaction may be of higher priority than an older one.

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.

III. Theoretical Background

A. 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 relative deadline D_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.

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

Scheduling of tasks. On multiprocessor systems, two alternative paradigms for scheduling collections of tasks are considered: *partitioned* and *global* scheduling. For 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. 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:

$$\sum_{\tau_i \in \tau} \frac{C_i}{P_i} \leq m \tag{1}$$

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.

B. Transaction Model in RT-DBMS

In Real-Time DataBases Management Systems (RT-DBMS), real-time transactions are characterized by several parameters on the basis of which scheduling decisions are made. These parameters are summarized hereafter. We consider a transaction system T. Each transaction $T_j \in T$ is characterized by a set of parameters r_j, W_j, D_j which respectively represent the transaction starting time, the worst-case execution time of T_j (i.e. the delay separating

the start time of T_j and its end time − considering equally both committed or aborted transactions − and the relative deadline of the transaction. Transaction T_j meets its deadline iff $W_j \in [r_j, r_j + D_j)$

Scheduling of 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.

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 the database consistency. In real-time database systems, not only database consistency should be satisfied, but transactions must also meet their deadlines [24]. Real-time concurrency control can be either *pessimistic* or *optimistic*. Pessimistic protocols are lock-based and systematically restrict all accesses to shared resources. For optimistic protocols, the detection and resolution of conflicts can happen after their occurrence. Intuitively, it seems that optimistic protocols have better performance. However, this is not easy to verify since their performance depends on several parameters [25].
To our knowledge, no real-time concurrency control policies are specially designed for software transactional memories. Furthermore, the real-time concurrency controls used in real-time database systems (see [26] for survey) are not suited for multiprocessors [27].

C. Transaction Model in STM

If we refer to the transaction model defined above for real-time databases, most of the existing STMs support only non real-time transactions (i.e. transactions without deadlines) defined as follows:

$$T_j = (r_j, W_j) \tag{2}$$

Current STM implementations usually use these parameters to set up the priority of transactions. This priority is used to resolve conflicts between transactions when they occur. Typically, policies like those implemented by the *Timestamp* or *Polite* contention managers use respectively the arrival time r_j of the transaction and the number n of retries (bounded by $\sum_n W_{j,n}$ units of time), to make their decisions [10].

D. STM implementations

1) Fraser's STM: FSTM [28] 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 transaction 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.
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 will 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).

In FSTM, the recursive helping is not systematically performed by a transaction during the attempt to commit. Only the transaction that is in the first (namely *write phase*) or in the second commit phase (namely *read phase*) can invoke the recursive helping. Moreover, the following conditions must be fulfilled:

- Let a transaction T_1 be in its write phase and attempting to acquire ownership of the object O. If O is owned by another transaction T_2 then T_1 will help T_2.
- Let both transactions T_1 and T_2 be in their read phase, and $\prec$ be a well-founded total order[6] on incomplete transactions. T_1 will abort T_2 iff $T_1 \prec T_2$. Otherwise T_1 will help T_2.

Note that a transaction in its read phase will never help a transaction in its write phase. Furthermore, imposing $T_1 \prec T_2$ guarantees that every cycle will be broken (*i.e.* a situation in which all transactions are in their read phase and try to read an object that is currently owned by the next transaction in the in-order queue).

2) Ennals' STM: Ennals' STM [5] is a lock-free-based STM. Unlike FSTM, each object is stored in place and there is no indirection to access the object. Each transaction maintains separate read and write descriptors for the opened objects, for reading and writing respectively. To write an

[3]System cannot tolerate the missing of deadlines.
[4]The system could accept the transaction even if it misses its deadline.
[5]Missing the deadline causes to abort the transaction.

[6]The relation $\prec$ is concretely implemented as ordering of the transaction descriptors.

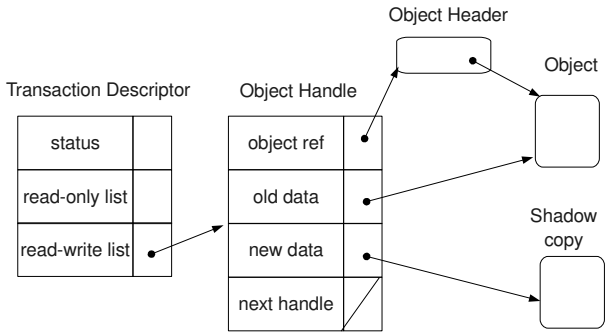

Fig. 1. Fraser's STM data structures

object, the transaction must first obtain an exclusive lock on the object so as to be able to create a working copy of the object. This lock is released only at commit or abort time. To read from an object, the transaction waits until the object's handle has a version number. Therefore, Ennals' STM uses a revocable two-phase locking scheme [29] for writings and an optimistic concurrency control for reads.

3) DSTM: Herlihy *et al.* propose the Dynamic STM [19]. The DSTM is a an obstruction-free-based STM which implements the weakest natural non-blocking progress guarantee. That is to say, at any point, a single process executed in isolation (i.e. with all obstructing processes suspended) for a bounded number of steps will complete its operation. In DSTM the transaction references an object through a *TMObject* structure that consists of a pointer to one and only one *Locator*. The Locator contains the reference to the descriptor of the transaction that created it. The old and the new version of the object are also contained in the Locator. The transaction descriptor has a read set to store all the objects accessed by the transaction. To acquire an accessed object, a new private locator is created with a cloned object. If the commit completes successfully, the read set is validated and the TMObject structure will be atomically updated with the new locator. In case of conflict, the DSTM does not set constraints on the selection of the transaction that should abort. Instead, it provides a general interface that allows a contention manager to implement a wide variety of policies [10].

IV. Real-Time STM

A. Real-time Transaction Model for STM

The classical model used so far to describe a transaction within STMs (see Equation 2) is not suitable for the real-time context. Indeed, in order to make scheduling decisions when conflicts occur between real-rime transactions, we need at least another parameter depicting the real-time constraints. Thus, based on both the transaction model found in RT-DBMS and that of real-time tasks, we consider a real-time transaction system T on $m \geq 1$ processors. Each transaction $T_j \in T$ is characterized by a set of parameters r_j, W_j, D_j which respectively represent the transaction starting time, the worst case execution time of T_j between r_j and the commit[7] time, and the relative deadline of the transaction. We define also the parameter $|D_j| = r_j + D_j$ which is the absolute deadline of the transaction T_j. The processor utilization for a transaction T_j on processor m is defined by $u_{j,m}$. Therefore, a real-time transaction in STM is defined as:

$$T_j = (r_j, W_j, D_j) \tag{3}$$

Transaction T_j meets its deadline iff $W_j \leq D_j$

B. RT-STM Description

Our *Real-Time Software Transaction Memory* (RT-STM) is an enhancement of Fraser's STM to take into consideration real-time transactions based on the model previously introduced. First, let us argue about the reasons why we chose Fraser's STM for our implementation. Considering our real-time transaction model, the challenge was to improve the total number of transactions that meet their deadlines. For that purpose, we opted for an optimistic protocol for resolving conflicts, because we believe that one of the basic ideas of transactional memories is to avoid the systematic blocking inherent to pessimistic protocols. Moreover, for soft real-time applications, we aim to avoid the situations in which a low-priority transaction blocks a higher-priority one. In that sense, the helping mechanism between transactions is an important notion in a real-time context. Indeed, a transaction with a low priority can help a transaction with higher priority. With respect to progress guarantees, lock-free-based algorithms seem more suitable for our purposes since they ensure that at least one transaction will make progress. For the reasons exposed above, Fraser's STM appears to be the best candidate for supporting soft real-time transactions.

C. Real-time Scheduling Rules

Hereafter are presented the scheduling rules that apply when conflicts occur between concurrent soft real-time transactions in RT-STM:

- Let a transaction T_1 be in its write phase and attempting to acquire ownership of the object O. If O is owned by another transaction T_2 and $|D_1| > |D_2|$ then T_1 will help T_2.

- Let both transactions T_1 and T_2 be in their read phase. T_1 will abort T_2 iff $|D_1| \leq |D_2|$. Otherwise, T_1 will help T_2.

These rules consist in a modification of the conditions of helping found in FSTM. The condition that $\prec$ is a well-founded total order defined for FSTM [28] is also fulfilled here since $|D_j|$ are chronologically ordered.
By imposing these conditions, transactions will be dynamically scheduled according to their deadlines, and only the transactions which have a higher-priority (*i.e* close to their deadlines)

[7]In this study, we focus only on transactions that meet their deadlines. Transactions rollback times will be considered in future work.

will be helped. Note that these modifications concern only the commit procedure in its read and write phases. In fact, unlike real-time database systems in which the ACI properties are usually relaxed to serve the real-time constraints of transactions, in our case, data are just as important as the deadlines of transactions. The read and write procedure objects still remain unchanged.

D. Implementation details

Intuitively, to ensure that transactions will meet their deadlines in RT-STM, the underlying operating system (OS) has also to be considered since transactions are executed within threads. Then, the OS must provide real-time constraints to transactions. That is why we used the real-time operating system (RTOS) named LITMUS^{RT} [8] [30]. Designed to run on top of a symmetric multiprocessor (SMP) architecture, it implements all the real-time task scheduling algorithms described in section III A. 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 our RT-STM on multiprocessor systems, under a panel of advanced real-time scheduling policies.

We have modified Fraser's STM and then integrated it into the LITMUS^{RT} library, as described below.

1) FSTM Modifications: On one hand, we have added a scheduling information into the transaction context in order to support soft real-time transactions. These informations are grouped together within a structure called *RTSched* and result from our real-time transaction model. At initialisation time, D_j is given as an input parameter for the transaction and *RTSched* is initialised with both the current tick value of the processor r_j and $|D_j|$. (see Algorithm 1).

Algorithm 1 Init Real-Time Transaction T_j

Require: D_j
$T_j.RTSched_j.r_j \Leftarrow ReadProcessorTicks()$
$T_j.RTSched_j.|D_j| \Leftarrow RTSched.r_j + D_j$

On the other hand, we have implemented the real-time scheduling rules presented in Section IV C. within the recursive commit function (for readibility, the algorithm that follows the given rules is not presented here).

2) 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 initialises 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.

[8] http://www.cs.unc.edu/∼anderson/litmus-rt

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. Our experiments under the resulting STM, namely RT-STM which rely on this hybrid scheme, are described in the next section.

V. Experimental Evaluation

We present here the experiments we performed to evaluate our RT-STM in terms of deadline guarantees for transactions. 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 STM among FSTM, DSTM and Ennals' STM, under Linux and LITMUS^{RT} operating systems. Then, we study the scalability of STMs under different real-time task scheduling policies. Finally, we evaluate our RT-STM proposal with respect to the best STM for our purpose.

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 [28] 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 also considers 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 and 25% of writes and deletes. 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 evaluation. These are described below.

Worst-case execution time jitters. We define the worst-case execution time of the set T by $WCET = \max\{W_j\}$. In our case, WCET jitters are computed as the difference between max $\{WCET\}$ and min $\{WCET\}$ for 10 experiments. Indeed, after 10 experiments, we note that the value of the WCET jitters remains in the range delimited by its previous min and max values.

In order to extract the W_j value, we have modified the different STMs. In fact, at commit time, we recover the current processor ticks by calling the assembly instruction *rdtsc*. The W_j of the transaction is thus obtained by subtracting the r_j parameter to the successful commit time. However, this method to get the ticks value at user-level is technically aberrant. Indeed, if transaction T_j starts on one core and migrates on other core where it commits, 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 2) which consists in adding the core identity to the context of the transaction. This is done by calling the assembly instruction *cpuid*[9]. 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 (line 9) or consider the test as a bad one at commit time (line 11). At the end of the experiment, if the number of transactions that have experimeted 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 2. 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, plus 2×0.5 at the commit time). Therefore, W_j has a precision within the interval $[1, 2]\mu$s.

Algorithm 2 W_j measurement

```
1:  init RetryCPU ⇐ 2
2:  T_j.coreID ⇐ CPUID()
3:  repeat
4:      RetryCPU ⇐ RetryCPU − 1
5:      T_j.RTSched_j.r_j ⇐ ReadProcessorTicks()
6:  until T_j.coreID = CPUID() Or RetryCPU = 0
7:  if RetryCPU = 0 then
8:      if state = TransactionStarting then
9:          Abort()
10:     else
11:         BadTest ⇐ BadTest + 1
12:     end if
13: end if
```

A. STMs' scalability under Linux and LITMUS^{RT}

In this experiment, we intent to show how the underlying operating system can impact on the variation of the W_j parameters, namely the *WCET jitters*. To cover the three main categories of STM, we have compared the Fraser's STM with the lock-based STM due to Ennals and with the obstruction-free-based STM due to Herlihy *et al.*

1) Experiment results: **Under Linux.** Fig. 2. shows that Ennals' STM does not scale. Ennals' STM suffers from frequent transaction collisions, and the transaction can wait for a long time before having access to the concurrent resources. This is due to two reasons. Firstly, the transaction is blocked during the commit time when its resources are owned by another transaction. Secondly, Ennals's STM places a restriction on the number of transactions which cannot exceed the number of cores at any time. This restriction is made in order to fully use all the cores [5]. This result about Ennals' STM confirms the results obtained in [16] in which Ennals's algorithm also performs badly. Therefore, Ennals' STM has not been taken into consideration for the rest of the experiments.

Under LITMUS^{RT}. Fig. 3. shows that both FSTM and DSTM behave better under a real-time operating system. As expected, the WCET jitters are more important under Linux due to preemption times caused by the interference of other applications. Under LITMUS^{RT}, on the contrary, the threads that execute our test have the greatest priority, and cannot be preempted by any Linux process.

B. STMs' scalability under RT task scheduling policies

As both FSTM and DSTM scale better under a real-time environment, the rest of experiments is thus conducted only with real-time task scheduling policies in order to determinate for which policy the STMs scale better.

1) Experiment results: **Under Pfair.** Fig. 4. and 5. show that both for the FSTM and DSTM, the Pfair policy produces the worse performance. This result can be explained by the fact that the Pfair scheduling policy is more complex than EDF-based approaches, thus involving more important scheduling overheads.

Under G-EDF. Unlike FSTM, the DSTM scales pretty well under this policy while the number of threads does not exceed 8. In this case, the high jitters observed under FSTM are caused by the extra bookkeeping information. The data in the stack are then more important in FSTM. Consequently, the migration cost with G-EDF is more important and causes more overheads in FSTM than in DSTM. Nevertheless, beyond 8 threads, unlike DSTM, FSTM keeps its scalability and the WCET jitters are relatively deadened.

Under P-EDF. FSTM outperforms DSTM and P-EDF is revealed as the best policy. Indeed, there are no migrations cost and the overheads observed are lesser. Furthermore, the result

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

obtained here presents some similarities to the experiments conducted by Branderburg *et al* (see section II) and also confirms that lock-free algorithms scale well under P-EDF than G-EDF.
Therefore, the rest of the experiments are conducted only under FSTM with the P-EDF scheduling policy.

C. Evaluation of our RT-STM

We now present, the comparison between RT-STM and a modified version of FSTM (to consider only transactions' deadlines), under P-EDF policy. We have studied the impact of both the number of threads and the variation of the deadline window length of soft real-time transactions, upon the system performance.

Deadline guarantee ratio. This parameter measures the number of transactions that successfully meet their deadlines at commit time over the total number of launched transactions. The deadline guarantee parameter has been integrated to both FSTM and RT-STM in order to perform the comparison. First, to take into consideration the deadlines of transactions, we integrated Algorithm 1 into FSTM. Afterwards, we modified the successful commit parts of the commit function in both FSTM and RT-STM. After being ensured that the transaction is running on its start-core (see Algorithm 2), the current processor ticks are compared to $|D_j|$ in order to verify whether the transaction has met its deadline or not. If this is the case, the number of transactions that meet their deadlines is then incremented atomically using the CAS instruction. In a similar way, the total number of transactions is atomically incremented at the startup time of each transaction.

The deadline window factor. Like in [25], for each transaction, a specific deadline is randomly (*rnd*) generated as follow :

$$D_j = rnd[0, k \times base) \tag{4}$$

where k represents the deadline window factor. The value of $base$ is fixed at 548 and is a processors frequency multiple.

1) Experiments results: **FSTM vs RT-STM.** Fig. 6. shows the absolute deadline guarantee ratio measured under FSTM and RT-STM. We observe that RT-STM outperforms FSTM. The outperformance ratio is constant and independent of the number of threads used.
Unlike in real-time databases in which transactions are usually large, in STMs the number of transactions is relatively more important, and W_j parameter is smaller since data are only memory-resident. Therefore, a small difference of the performance ratio between STMs involves a great number of transactions. For instance, the improvement of RT-STM over FSTM is about 10^5 of transactions that meet their deadlines during the 10 seconds of the test execution. The outperformance ratio remains constant even when increasing the test duration, but the total number of transactions increases. Thus, on the basis of the number of transactions, the outperformance of RT-STM will be more and more important, when increasing the test duration.

RT-STM benefits. Fig. 7. shows the relative deadline guarantee ratio of our RT-STM with respect to that of FSTM under varying deadline window lengths. When the generated deadlines are lesser than 548 processor ticks (i.e, $k \in [0, 1)$) transactions miss their deadlines both in FSTM and RT-STM since the deadline interval is very small. On the contrary, for $k \in [64, 128]$ we see that all the deadlines are met for the two STMs since the real-time constraints are easier to satisfy in this case. However, note that for $k \in [2, 4]$ the outperformance of our RT-STM is maximal. In fact, this maximum ratio corresponds to the situation in which the deadline guarantee ratio is equal to 50% both in FSTM and RT-STM. Furthermore, the maximum performance obtained in our RT-STM is essentially due to the first rule set for resolving conflicts (see Section IV). Indeed, the second rule of RT-STM rarely occurs and is defined only to prevent the read cycles.

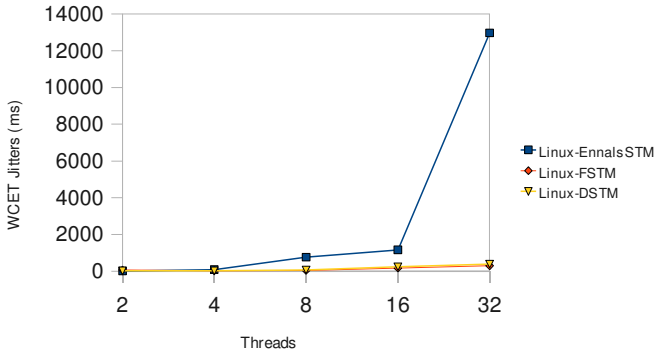

Fig. 2. STMs scalability under Linux

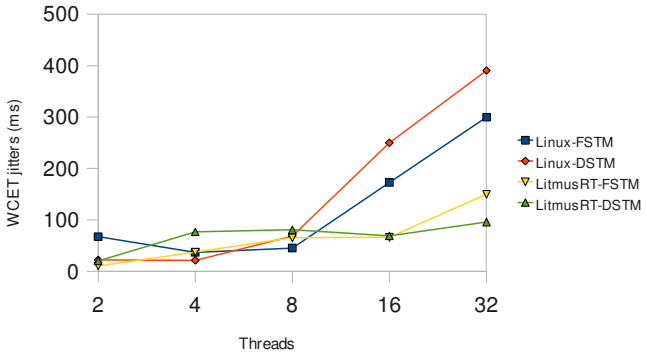

Fig. 3. STMs scalability under LITMUS^{RT}

VI. Conclusion

While real-time scheduling of transactions are widely studied in a real-time databases, the issue has not yet been addressed for transactional memories. Motivated by this observation, we introduced a real-time transaction model into software transactional memory and defined a new real-time scheduler of transactions based on the evaluation of deadlines. To our knowledge, such study has not been attempted before.

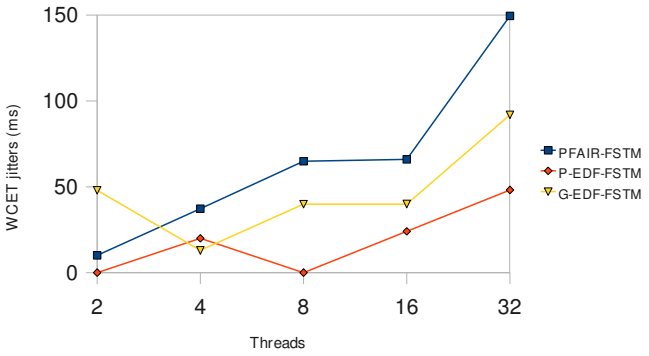

Fig. 4. FSTM scalability under RT scheduling policies

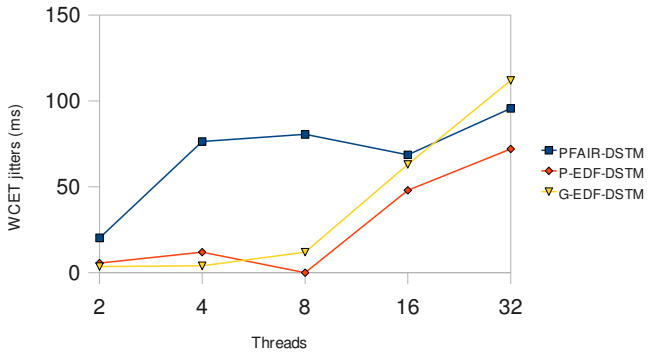

Fig. 5. DSTM scalability under RT scheduling policies

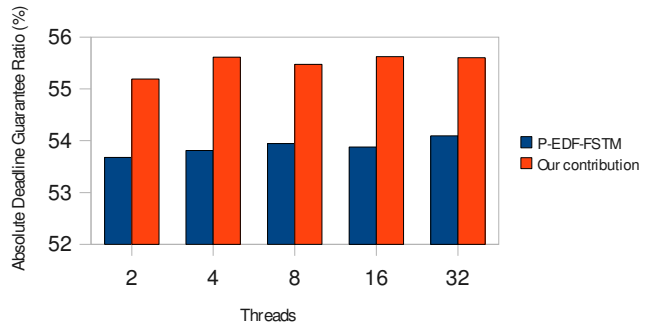

Fig. 6. FSTM vs RT-STM

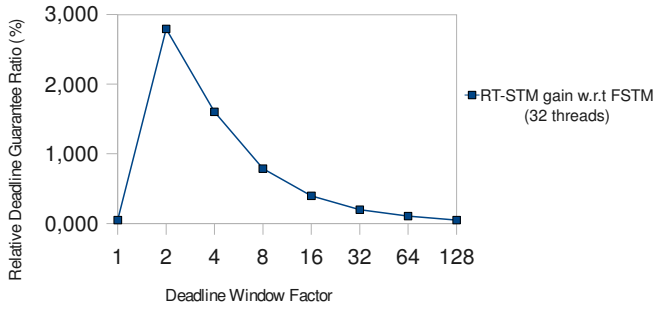

Fig. 7. RT-STM benefit

The main conclusions of our study can be summarized as follows: **(i)** Using the LITMUS^{RT} RTOS, the execution time of transactions has less jitters than under classical (i.e. non real-time) OS, both for FSTM and DSTM; **(ii)** Using P-EDF, both FSTM and DSTM scale better than under other real-time policies, namely G-EDF and Pfair; **(iii)** Using P-EDF, FSTM outperforms DSTM in terms of execution time jitters; **(iv)** Using P-EDF, our contribution outperforms FSTM in terms of the number of transactions that meet their deadlines.

For future work, there are many possible directions. First, in our experiments we assumed a garbage collector (GC) provided by the FSTM implementation. We believe that a GC has a great influence on the execution time of the transactions. One optimization could be dedicated to the design of a specific real-time task for GC. Second, in our experiments, we arbitrarily fixed the parameters of the real-time tasks. It would be 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] J. Huang and J. Stankovic, "Concurrency control in real-time database system : Optimistic scheme vs. two-phase locking," UM-CS-1990-066, University of Massachusetts, Tech. Rep., 1990.

[26] U. Shanker, M. Misra, and A. K. Sarje, "Distributed real time database systems: background and literature review." *Distributed and Parallel Databases*, vol. 23, no. 2, pp. 127–149, 2008.

[27] T. W. Kuo, J. Wu, and H. C. Hsih, "Real-time concurrency control in a multiprocessor environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 6, pp. 659–671, 2002.

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

[29] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system," *Communication of the ACM*, vol. 19, no. 11, pp. 624–630, 1976.

[30] 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.