IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Managing contention of software transactional memory in real-time systems

António Barros

Luís Miguel Pinho

HURRAY-TR-101102

Version:

Date: 11-30-2010

Managing contention of software transactional memory in real-time systems

António Barros, Luís Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

The foreseen evolution of chip architectures to higher number of, heterogeneous, cores, with non-uniform memory and non-coherent caches, brings renewed attention to the use of Software Transactional Memory (STM) as an alternative to lock-based synchronisation. However, STM relies on the possibility of aborting conflicting transactions to maintain data consistency, which impacts on the responsiveness and timing guarantees required by real-time systems. In these systems, contention delays must be (efficiently) limited so that the response times of tasks executing transactions are upper- bounded and task sets can be feasibly scheduled. In this paper we defend the role of the transaction contention manager to reduce the number of transaction retries and to help the real-time scheduler assuring schedulability. For such purpose, the contention management policy should be aware of on-line scheduling information.

Managing contention of software transactional memory in real-time systems

António Barros and Luís Miguel Pinho
CISTER Research Center, Polytechnic Institute of Porto, Portugal
{amb,lmp}@isep.ipp.pt

***Abstract*—The foreseen evolution of chip architectures to higher number of, heterogeneous, cores, with non-uniform memory and non-coherent caches, brings renewed attention to the use of Software Transactional Memory (STM) as an alternative to lock-based synchronisation. However, STM relies on the possibility of aborting conflicting transactions to maintain data consistency, which impacts on the responsiveness and timing guarantees required by real-time systems. In these systems, contention delays must be (efficiently) limited so that the response times of tasks executing transactions are upper-bounded and task sets can be feasibly scheduled. In this paper we defend the role of the transaction contention manager to reduce the number of transaction retries and to help the real-time scheduler assuring schedulability. For such purpose, the contention management policy should be aware of on-line scheduling information.**

I. Introduction

Multicore processors bring new challenges to real-time systems (RTS), pushing research for alternative solutions more suited to the parallel execution capabilities provided by the new architectures. Although not yet critical, the tendency for even larger number of cores will further impact the way systems are developed[1]. In uniprocessors, lock-based solutions became commonplace as the means to avoid race conditions, despite the well-known pitfalls, such as complexity, lack of composability [2] or (bounded) priority inversion. In multiprocessor systems these solutions introduce additional drawbacks. Concurrency is impaired as locks serialise non-conflicting operations (which could actually be in parallel) on disjoint parts of a shared resource, and may cause cascading or convoying blocks [3], causing a severe impact on the system throughput.

An alternative explored in highly-parallel systems is the *non-blocking object*. There are no locks, so a task will not block when accessing resources. Conflicts are managed by an underlying mechanism that maintains consistency of the shared object. On multiprocessors, this solution presents strong conceptual advantages [4] and has been shown in several cases to perform better than lock-based ones [5], [6]. Priority inversion and deadlocks are eliminated because accesses to the object proceed in parallel and concurrent accesses *can be resolved in favour of higher-priority tasks*. The convoy effect is eliminated because no task will block waiting for a task that is failing or prevented from executing. These advantages are impaired by the complexity to maintain the consistency of the shared object. This complexity depends on the progress guarantees provided [7]. *Wait-free* assures that every call will finish in a finite number of steps independently on the pace other tasks execute, so all tasks are assured to progress. *Lock-free* relaxes this condition, so that the system is able to progress, although some tasks may starve. The most relaxed guarantee of progress is given by *obstruction-free*, which assures that one task will eventually progress if executes isolated from other concurrent tasks.

Still, the consistency of the object may not depend on a single operation but on a composition of operations, eventually involving other shared objects, as if executed atomically in a *transaction* in shared memory space: *transactional memory* [8]. Shavit and Touitou [9] later on adapted the concept, implementing all operations totally on software: the *Software Transactional Memory* (STM). This first approach supported static transactions, which required transactions and memory usage to be defined in advance. More recent implementations of STM (*e.g.* [10], [11]) provide dynamical transactions (a transaction can decide which addresses to access based on values read at run-time) and dynamical memory usage, with diverse synchronisation mechanisms.

A transaction is executed sequentially in isolation, regardless of other parallel transactions and it must complete, either succeeding (*commit*) or failing (*abort*). Consistency is maintained as transactions operate on private copies of public data. Before completing, all accessed locations are checked for conflicting updates that may have occurred. If no conflicts are detected then the data is consistent and updates become effective. When conflicts are detected, a contention policy is applied in order to allow, at least, one transaction to successfully commit. This policy dictates how contending transactions should behave to provide the progress guarantees and transaction prioritisation. Typically, when a transaction detects another transaction accessing common memory addresses, it may choose to abort the contender or to give it the opportunity to conclude, depending on particular criteria. The contention policy must avoid *livelock*, so a pair of conflicting transactions should not be able to indefinitely abort each other. The STM concept is more relevant in multiprocessors, and it becomes an (non-exclusive) alternative to lock-based synchronisation when a high-degree of parallelism with low contention is available.

[1] E.g., the experimental Intel Single-chip Cloud Computer (SCC) [1] carries 48 cores, has message-based interconnection and no cache coherency

Nevertheless, in RTS jobs have to execute considering their deadlines, so the contention manager must be responsible to bound the delay that any transaction might suffer, to avoid deadlines being missed. Furthermore, although STM may improve responsiveness in low-contention situations, we still need to analise worst-case response times. Contention management techniques that have shown to improve system progress (such as adaptive exponential back-off) may not suit the timing requirements of RTS.

II. Related work

Although the concept of STM is not new, only a few works dealt with it in the context of real-time systems. In [12], a data access mechanism is proposed for uniprocessor platforms – the Preemptible Atomic Regions – together with an analysis to bound the response time of jobs. An atomic region is guaranteed to be free from other tasks' interference because any transaction pre-empted by a higher-priority task is immediately aborted, and its effects undone. This policy implies that no concurrent transactions are allowed in the system, and it is impractical in multiprocessor systems.

In [13], and based on previous work on lock-free objects, scheduling conditions are established for lock-free transactions under EDF and DM; however, this work is only valid for uniprocessor systems. For the case of multiprocessors, a different approach to support transactions is provided [14]: a wait-free mechanism relying on a helping scheme, which provides an upper bound on the transaction execution time. This approach is nevertheless pessimistic, since an arriving transaction must help pending transactions before being able to proceed, even if no conflicts would occur; additionally, the upper bound depends on the number of processors so it will not scale with increasing number of cores.

In [15], an algorithm is described to calculate an upper-bound on the worst-case response time of tasks on a multiprocessor system using STM. Tasks are scheduled with the Pfair approach. Each task can have multiple atomic regions, and concurrent transactions can interfere with each other. Conflicts are detected and solved during the commit phase. This analysis is limited to small atomic regions, assuming that any transaction will execute at most in two quanta.

In [16], real-time scheduling of concurrent transactions is proposed for soft real-time. In the proposed model, transactions are characterised by scheduling parameters, which are taken into account whenever solving a detected conflict between transactions. However, the policy to serialise transactions is based on the absolute deadlines of the contending transactions, which can lead to missed deadlines.

Finally, [17] proposes hardware transactional memory. It assumes each task contains one single atomic region, and conflicts are detected and solved during the commit phase. The analysis demonstrates that the jobs of a task will meet their deadlines as long as the transactions of two consecutive jobs are separated by the resolve time, the worst-case time a transaction will take to successfully commit. Nevertheless, the paper does not describe any method to solve transaction conflicts based on on-line scheduling data.

These works provide already some perspectives on how to deal with STM in real-time systems. However, it is clear that there are many issues pending, and further research is necessary to take advantage of future parallel architectures. Therefore, this paper proposes new approaches to manage contention between conflicting transactions, using on-line information, with the purpose of reducing the overall number of retries, increasing responsiveness and reducing wasted processor utilization, while assuring deadlines are met.

III. Managing contention in the real-time domain

A. System model

The system model assumes that jobs are released by a set of periodic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ and scheduled on m identical processors. Each task τ_i is characterised by (T_i, C_i); T_i is the period of job arrivals and C_i the worst-case execution time (tasks have implicit deadlines). The jth job of task τ_i, hence forward denominated J_{ij}, is characterised by (r_{ij}, d_{ij}), being r_{ij} the time the job is released and d_{ij} the absolute deadline of the job, defined as

$$d_{ij} = r_{ij} + T_i. \tag{1}$$

In this initial analysis, each job is assumed to perform at most one transaction, which may be aborted multiple times; every time it is aborted, it restarts immediately. A transaction is characterised by W_i, its maximum execution time when no contention occurs. A transaction can fail during the commit phase, if it finds conflicting data accesses. Also, STM objects are assumed to be globally accessible to tasks, independently of the processor in which transactions are executing, and multiple simultaneous transactions are supported.

In uniprocessor lock-based solutions, the blocking delay is the time taken since a job requests the lock until the access is granted; during this time interval, the resource is granted to lower-priority jobs; once the lock is acquired, the job is guaranteed to complete the critical section. In such cases, blocking delays are bounded by the number of lower-priority jobs that block the execution of a critical section [18], [19]. When data is shared using STM, delay has a totally different nature. When a transaction starts, it may successfully complete if no data access conflicts occur, finishing with no delay. But if a conflict occurs, the transaction may be aborted and will have to restart from the beginning. If the transaction is aborted multiple times, the delay will grow linearly with the number of unsuccessful completions. It becomes obvious that bounding the delay of a transaction is equivalent to limiting the number of times the transaction may be aborted.

The response time of a job is here defined as the time elapsed since a job arrives until its execution is completed.

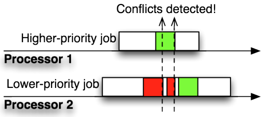

Figure 1. Lower-priority task aborted by a higher-priority transaction.

For a job release of task τ_i that executes a transaction, the response time RT_i depends on the execution time of the task, the interference time I_{ij} in which the job was preempted by higher-priority tasks and the overhead due to the n aborted executions of the transaction. The response time of a job can be defined by

$$RT_{ij} = C_i + I_{ij} + n \cdot W_i \tag{2}$$

and to meet deadlines, the response time must be

$$RT_{ij} \leq D_i \tag{3}$$

From equations (2) and (3) we can conclude that the amount of times the transaction can be aborted depends on the slack time available to the job.

B. Managing contention in fixed-priority scheduling

In this analysis, it is assumed that priorities are the same across processors, thus the following considerations are valid for both partitioned fixed-priorities scheduling and global fixed-priorities scheduling.

Since priorities are used to somehow express the urgency, even if not necessarily the importance, of a task, it is quite natural to give higher priority to transactions performed by higher priority jobs. Therefore, any contention manager in a real-time system should use task priorities to decide which transaction should proceed, in case of conflicts. Figure 1 represents two jobs executing on two processors, with transactions that access common data objects; in this example it is assumed that conflicts are detected during the commit phase. When the lower-priority transaction commits, it fails in favour of the higher-priority task, resuming immediately after; when the higher-priority transaction commits, it succeeds and the lower-priority transaction is aborted; finally the lower-priority transaction succeeds on the third try.

Considering the priority as the sole factor in resolving a conflict is simple and fast, but has inconvenient drawbacks. In the example, the lower-priority job has its response time increased, as the transaction is retried. If the transaction is aborted several times due to incessant conflicting higher priority transactions, then the job might not be able to complete within the deadline. Another case is when a higher-priority transaction becomes unsound, lower-priority transactions will never succeed until the unsound transaction is aborted, similar to the convoying effect. Transaction starvation can be avoided if additional on-line information is used by the contention manager, which is explored in the next two algorithms.

```
if $this.priority \geq contender.priority$ or $this.can\_abort() = FALSE$ then
    abort $contender$
else
    abort $this$
end if
```

Figure 2. Deciding algorithm that avoids starvation.

```
if $this.can\_abort() = FALSE$ then
    abort $contender$
else
    if $contender.can\_abort() = FALSE$ then
        abort $this$
    else
        if $this.retries < contender.retries$ and $this.priority < contender.priority$ then
            abort $this$
        else
            abort $contender$
        end if
    end if
end if
```

Figure 3. Deciding algorithm aware of number of retries.

1) An algorithm that avoids starvation: The algorithm in Figure 2 is executed when a transaction is in the commit phase, and decides the conflict in favour of the higher-priority transaction, except if the committing transaction has lower-priority but the analysis of its slack time performed by $can\_abort()$ function reveals that the job will miss the deadline if the transaction is aborted. The slack time can be calculated as follows

$$slack_{ij} = (d_{ij} - clock) - c_i^{bt} \tag{4}$$

in which c_i^{bt} is the remaining execution time required by the job before initiating the transaction.

Although this algorithm allows one lower-priority transaction to be consistently aborted by contending transactions with higher-priorities, it does impose a limit on the number of retries, guaranteeing individual job progress.

2) An algorithm aware of retries: Reducing the number of times a transaction is aborted can be achieved if each transaction has a counter that is incremented each time the transaction fails. Knowing the number of failed attempts allows us to build deciding algorithms that are able to detect and react on excessively aborted transactions.

The algorithm presented in Figure 3 has three levels of decision. The first prevents the commiting transaction to be aborted if it has no slack to repeat. The second does the same for the contending transaction. The final level combines priorities and number of retries to decide which transaction should abort, observing task priorities but simultaneously favouring a transaction ordering based on the chronological arrival of commits, allowing lower-priority tasks to abort higher-priority tasks as long as no deadlines are missed.

```
if $this.can\_abort() = FALSE$ or $contender.can\_abort() = TRUE$ then
    abort $contender$
else
    abort $this$
end if
```

Figure 4. Deciding algorithm that favours committing transaction.

This algorithm distributes the number of transaction aborts among the tasks, across priorities, avoiding the concentration of retries in certain tasks and the consequent abnormal increase of their rates of processor utilisation. An interesting effect is that short lower-priority transactions will be able to succeed sooner at the expense of aborting a longer contending transaction with higher-priority; this approach eliminates the negative effects of unsound transactions over lower-priority contenders.

C. Managing contention in EDF-based scheduling

Similar to the previous analysis of STM contention management on fixed-priorities scheduling, we will consider jobs are scheduled by the system according to their deadlines, and the contention manager does not discriminate the processors in which transactions are being executed. Thus, the job with the nearest deadline is the most urgent, independently of the processor it is assigned.

In EDF, the contention manager can always choose in favour of the transaction being executed by the job with the closest deadline. However, under heavy contention, jobs with relatively far deadlines can have their transactions aborted several times before succeeding, increasing abnormally their execution times and, as a consequence, increasing the processor utilization. Similar to a job spin-locking, the job executing the repeatedly-failed transaction would not produce effective work, while not yielding the processor to another ready job. Additionally, a job with farther deadline but short slack can miss its deadline if the transaction is aborted several times.

The algorithm in Figure 4 favours the committing transaction, preserving any concluded work, as long as none of the involved jobs will miss their deadlines.

IV. Future work

Software Transactional Memory is the subject of much research in parallel and distributed systems in recent years. The concept exploits optimistic operations on data, allowing disjoint-access parallelism, thus being appealing in multiprocessor systems. This paper explores the role of the contention manager to help the real-time scheduler assuring the schedulability of concurrent task sets. Two algorithms for fixed-priority scheduling and one for EDF-based scheduling are presented. We are currently working on the response time analysis for the proposed contention management approaches in order to determine upper bounds for the number of retries. As future work, we intend to explore the trade-offs between early and late contention and suspension-based solutions, and analyse the effects of transaction management in more dynamic systems, considering bandwidth server-based scheduling.

Acknowledgment

The authors would like to thank Björn Andersson for comments on a previous version of this work.

This work was supported by FCT through the CooperatES (PTDC/EIA/71624/2006) and RESCUE (PTDC/EIA/65862/2006) projects, and by the European Commission through the ARTIST2 NoE (IST-2001-34820).

References

[1] "The SCC Platform Overview," Intel Labs, Santa Clara, CA, USA, Tech. Rep., May 2010. [Online]. Available: http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf

[2] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[3] B. N. Bershad, "Practical considerations for non-blocking concurrent objects," in *ICDCS 1993.*, May 1993, pp. 264–273.

[4] P. Tsigas and Y. Zhang, "Non-blocking data sharing in multiprocessor real-time systems," in *RTCSA'99*, Dec. 1999, pp. 247–254.

[5] 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 *RTAS '08.*, Apr. 2008, pp. 342–353.

[6] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui, "Why STM can be more than a Research Toy," *accepted for publication, CACM*, 2010.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[8] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *ISCA '93*, vol. 21, no. 2, May 1993, pp. 289–300.

[9] N. Shavit and D. Touitou, "Software transactional memory," in *PODC '95*, vol. 22, no. 3, Aug. 1995, pp. 204–213.

[10] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, Sep. 2003. [Online]. Available: http://www-test.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[11] R. Ennals, "Efficient Software Transactional Memory," Intel Research Cambridge, Cambridge, UK, Tech. Rep., 2005. [Online]. Available: http://berkeley.intel-research.net/rennals/pubs/051RobEnnals.pdf

[12] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek, "Preemptible Atomic Regions for Real-Time Java," in *RTSS'05*, Dec. 2005, pp. 62–71.

[13] J. H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay, "Lock-free transactions for real-time systems," in *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, May 1997, pp. 215–234.

[14] J. H. Anderson, R. Jain, and S. Ramamurthy, "Implementing hard real-time transactions on multiprocessors," in *Real-Time Database and Information Systems: Research Advances*. Kluwer Academic Publishers, Sep. 1997, pp. 247–260.

[15] S. F. Fahmy, B. Ravindran, and E. D. Jensen, "On Bounding Response Times under Software Transactional Memory in Distributed Multiprocessor Real-Time Systems," in *DATE '09*, pp. 688–693.

[16] T. Sarni, A. Queudet, and P. Valduriez, "Real-Time Support for Software Transactional Memory," in *RTCSA' 2009*, pp. 477–485.

[17] M. Schoeberl, F. Brandner, and J. Vitek, "RTTM: Real-Time Transactional Memory," in *SAC '10*, Mar. 2010, pp. 326–333.

[18] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, Sep. 1990.

[19] T. P. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, Mar. 1991.