Real-time scheduling of transactions in multicore systems

Toufik Sarni, Audrey Queudet, Patrick Valduriez

Real-time scheduling of transactions in multicore systems

Toufik Sarni, Audrey Queudet, Patrick Valduriez
emails: {firstname.name@univ-nantes.fr, Patrick.Valduriez@inria.fr}

December 18, 2008

Abstract

Transactional memory has attracted much interest for multicore systems as it eases programming and avoids the problems of lock-based methods. However, introducing real-time scheduling of transactions in multicore systems is an open problem. Existing solutions for real-time scheduling consider either tasks in multiprocessor systems or transactions in database systems. In this paper, we show that these solutions are not suitable for multicore systems. And we discuss the main challenges to introduce real-time scheduling within transactional memory in multicore systems.

1 Introduction

With the advent of multicore systems, the transactional memory concept has attracted much interest from both academy [7, 13] and industry [14] as it eases programming and avoids the problems of lock-based methods. By supporting the ACI (Atomicity, Consistency and Isolation) properties of transactions, transactional memory 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. Furthermore, in the case of multicore systems, lock-based synchronization can reduce the data bandwidth by blocking several processes that try to access critical sections, thus reducing processors utilization. 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.
However, real-time scheduling of transactions, which is needed for many real-time applications, is an open problem in multicore systems. Existing solutions for real-time scheduling of either tasks in multiprocessor systems or transactions in database systems are not suitable for multicore systems. Real-time scheduling of tasks in multiprocessor systems does not consider important features of multicore systems, such that the presence of on-chip shared caches. Real-time scheduling of transactions in database systems has been around since the 80s but assuming either centralized or distributed systems. Thus, the solutions are not suitable for multicore systems as well.

In this paper, we briefly review these main solutions and discuss the main challenges to introduce real-time scheduling within transactional memory in multicore systems. The rest of the paper is organized as follows. Section 2 reviews real-time task scheduling in multiprocessor systems. Section 3 reviews real-time transaction scheduling in database systems. Finally, Section 4 discusses the main challenges for real-time transactional memory.

2 Multiprocessor real-time task scheduling

Context. We consider the scheduling of a task system on multiprocessors. For each task, a set of jobs is associated. At any time, each processor executes at most one job. The task has a *period* and an *execution requirement*. When a job is released, it executes during the execution requirement of the task, and once the period is elapsed, an other job of the task, is released.

Classification. In multiprocessor systems, two alternative paradigms for scheduling collections of tasks have been considered: *partitioned* and *global* scheduling. For the partitioned approach, the tasks are statically assigned to processors and then 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 to other processors is not allowed. Under the global scheduling approach, migrations are only allowed at job boundaries. A single queue and only one policy is applied to tasks.

Real-time scheduling algorithms. The classification for uniprocessor systems is usually based on the priority (static or dynamic) assigned to tasks. For preemptive uniprocessor systems, Earliest Deadline First (EDF) is optimal [11]. Unfortunately, EDF is *not* optimal on multiprocessors either under the partitioned or the global approaches [5], called respectively P-EDF and G-EDF. Another class of scheduling algorithms, which differs from the previous ones, is the *Pfair* algorithm [2]. It is based on the idea of *proportionate fairness* and ensures 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 executes within the associated window and migration is allowed for each subtask. PD^2 [10] is an optimal scheduling variant of Pfair.

Scheduling with shared resources. The protocols managing resources in real-time systems are usually used in a hard real-time context, such as M-PCP and FMLP[1][4] under EDF. For Pfair scheduling, a lock-free algorithm has been proposed [10] to ensure that *some* task is always making progress. Indeed, classical lock-based algorithms cannot satisfy this property.

What about multicore? To our knowledge, only one recent paper [5] deals with the *scalability* of the scheduling algorithms presented above, on multicore platforms[2]. One main conclusion of the authors is that on multicore platforms with on-chip shared caches, both the small size of the caches and the memory

[1] FMLP is a locking-based synchronization mechanism adapted both for P-EDF and G-EDF
[2] The study was conducted using the SUN-Niagara platform with 32 logical CPUs.

bandwith have negative impact on the algorithms, allowing migrations (*i.e* G-EDF, PD^2). Furthermore, for the global approach, the scheduling overheads greatly depend on the way of implementing the run queues. On the other hand, without resource sharing, P-EDF performs well for this study.
This means that *pure* global algorithms will not scale, and thus real-time global policies need to be revisited for many-core architectures. More particularly, the scheduler should be able to control more precisely the sharing of processor's internal resources (i.e. cache levels) by real-time tasks.

3 Real-time transactions scheduling in databases

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.

Real-time concurrency control. The scheduler of transactions in database systems, has a *concurrency control* protocol, which resolves conflicts between transactions when they occur, in order to maintain database consistency. In real-time database systems, not only database consistency should be satisfied, the transactions must meet their deadlines too[1]. Real-time concurrency control can be either *pessimistic* or *optimistic*. The pessimistic protocols systematically restrict all accesses to shared resources. For optimistic protocols, the detection and resolution of conflicts can happen after their occurrence. Many real-time concurrency protocols have been proposed (see [12] for survey) either pessimistic or optimistic, for centralized or distributed systems.
Intuitively, it seems that optimistic protocols have better performance. However, this is not be easy to verify since it depends on several parameters when doing comparison studies [9].

Pessimistic algorithms. Pessimistic algorithms use locking. The 2-PL-HP (*High Priority*) protocol [1] resolves the conflicts between transactions by considering their priorities. The priority of the transaction is based on its deadline. This protocol ensures that a high-priority transaction will not be blocked by a lower-priority transaction. Thus, it prevents deadlock situations.

Optimistic concurrency controls For firm real-time transactions, the $\binom{m}{k}$-firm [6] protocol considers m mandatory transactions and $k-m$ optional transactions. Mandatory (firm) transactions are aborted if they miss their deadlines. Optional transactions are executed only if there is no remainder mandatory transaction. The protocol tries to satisfy a maximum number of optional transactions to meet their deadlines. The Cost Conscious Approach (CCA) [8] improves 2PL-HP. The CCA protocol uses at run-time, the dynamic parameters of the transaction to reevaluate its priority. Some parameters are the rollback-times and the estimated remaining execution time. Speculative Concurrency Control (SCC) is an hybrid protocol [3] which combines the advantages of both pessimistic and optimistic protocols. When the conflict is detected in a trans-

[3]System cannot tolerate the missing of deadline, which can have catastrophic impacts.
[4]The system could accept the transaction even if it misses its deadline.
[5]Missing the deadline causes to abort the transaction.

action, a *shadow transaction* is created to "save" the conflict zone. Thus, the transaction can meet its deadline since only a proportion of itself (from the saved zone) is restarted. This protocol needs more resources than others since it duplicates both resources and the transactions when the conflicts occur.

What about transactional memory? A recent paper [15] deals with the scheduling of transactions for transactional memory. However, although it addresses the time aspect with an adaptive approach, the real-time scheduling of transactions is not considered and remain an open problem.
Although real-time concurrency control schemes provide a general framework for real-time transactions, actual implementations of these schemes can entail significant overhead, thus the interest of considering transactional memory where transactions are memory-resident data.

4 Discussion

Scheduling both tasks and transactions in multicore systems requires new policies that consider the multicore architecture features. Real-time scheduling of tasks for shared-memory multiprocessor systems is a good starting point but needs to carefully consider the presence of on-chip shared caches.
Real-time scheduling of transactions has been considered for uniprocessor or multiprocessor architectures, without shared memory (distributed systems) but needs to be tested on multicore platforms to evaluate its performance. Indeed, it is often claimed that the SCC protocol underperforms the other ones since it duplicates the resources and the transactions. But in a multicore system, with multi-threading technology, is this still true?
In distributed systems, the $\binom{m}{k}$-firm protocol outperforms SCC [6]. But does this comparison still hold for a multicore system at a single site?
If we consider the CCA protocol for real-time scheduling of transactions, what are the impacts of knowing the rollback-times and the remaining execution times of the transactions in multicore systems?
Addressing these main questions could help us formalizing the introduction of real-time scheduling of transactions within transactional memory.
Furthermore, with the advent of transactional multicore, the real-time scheduler of transactions could be integrated in hardware. It is then important to study the interactions between the schedulers of both tasks and transactions. Experiments similar to those presented in [5, 4] should be conducted in order to determine which real-time policy among (G-P)-EDF and PD^2, is more efficient in conjunction with the real-time scheduling of transactions. Thus, defining the utilization rate of the transactions on a given core can be helpful. For example, this rate can help the real-time task scheduler to promote the task with the smallest priority because having a transaction close to its deadline.

References

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

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

[3] Bestavros, A. Speculative concurrency control. Tech. rep., Boston University, Boston, MA, 1992.

[4] Brandenburg, B. B., Calandrino, J. M., Block, A., Leontyev, H., and Anderson, J. H. 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* (2008), IEEE Computer Society, pp. 342–353.

[5] Calandrino, B. B. J., and Anderson, J. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *proc. The 29th IEEE Real-Time Systems Symposium* (December 2008).

[6] Haubert, J., Sadeg, B., and Amanton, L. $\binom{m}{k}$-firm real-time distributed transactions. In *Proc. of the 16th WIP Euromicro Conference on Real-Time Systems (ECRTS)* (2004), pp. 61–65.

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

[8] Hong, D., Johnson, T., and Chakravarthy, S. Real-time transaction scheduling: a cost conscious approach. *SIGMOD Rec. 22*, 2 (1993), 197–206.

[9] Huang, J., and Stankovic, J. Concurrency control in real-time database system : Optimistic scheme vs. two-phase locking. Tech. rep., UM-CS-1990-066, University of Massachusetts, 1990.

[10] Leung, J. *Handbook of scheduling : algorithms, models, and performance analysis*. Chapman & Hall/CRC, 2004.

[11] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM 20*, 1 (1973), 46–61.

[12] Shanker, U., Misra, M., and Sarje, A. K. Distributed real time database systems: background and literature review. *Distributed and Parallel Databases 23*, 2 (2008), 127–149.

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

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

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