CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Non-preemptive scheduling of Real-Time Software Transactional Memory

António Barros
Luis Miguel Pinho

CISTER-TR-140207
Version:
Date: 2/17/2014

Non-preemptive scheduling of Real-Time Software Transactional Memory

António Barros, Luis Miguel Pinho

CISTER Research Unit

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: amb@isep.ipp.pt, lmp@isep.ipp.pt

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

Abstract

Recent embedded processor architectures containing multiple heterogeneous cores and non-coherent caches, bring renewed attention to the use of Software Transactional Memory (STM) as a building block for developing parallel applications. STM promises to ease concurrent and parallel software development, but relies on the possibility of abort conflicting transactions to maintain data consistency, which affects the execution time of tasks carrying transactions. Thus, execution time overheads resulting from aborts must be limited, otherwise the timing behaviour of the task set will not be predictable. In this paper we formalise a FIFO-based algorithm to order the sequence of commits of concurrent transactions. Furthermore, we propose and evaluate two non-preemptive scheduling strategies, in order to avoid transaction starvation.

Non-preemptive scheduling of Real-Time Software Transactional Memory

António Barros and Luís Miguel Pinho

CISTER/INESC-TEC
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
{amb,lmp}@isep.ipp.pt

Abstract. Recent embedded processor architectures containing multiple heterogeneous cores and non-coherent caches, bring renewed attention to the use of Software Transactional Memory (STM) as a building block for developing parallel applications. STM promises to ease concurrent and parallel software development, but relies on the possibility of abort conflicting transactions to maintain data consistency, which affects the execution time of tasks carrying transactions. Thus, execution time overheads resulting from aborts must be limited, otherwise the timing behaviour of the task set will not be predictable. In this paper we formalise a FIFO-based algorithm to order the sequence of commits of concurrent transactions. Furthermore, we propose and evaluate two non-preemptive scheduling strategies, in order to avoid transaction starvation.

1 Introduction

The current trend to increase processing power by manufacturing chips including multiple processor cores provided the ability to execute concurrent software in parallel. This tendency for even larger number of processor cores will further impact the way systems are developed. Some recently proposed architectures for embedded systems, like the STMicroelectronics P2012 [4] (prototypes are available with 69 cores), Kalray's MPPA [10] (up to 1024 cores; current version is 256 cores) allow both to concentrate multiple applications into the same processor, maximizing the hardware utilisation, and reducing cost, size, weight, and power requirements, and also to improve application performance by exploiting parallelism at the application level.

Nevertheless, integrating a high number of cores in a chip raises several problems, due to core interconnection and memory hierarchy. Cache coherency is being challenged [7] although some solutions can scale to dozens of cores [13], and some chips still provide (software-based) solutions. Buses do not to scale and the paradigm is shifting to networks-on-chip (NoC). Furthermore, platforms can be homogeneous, with either symmetric multiprocessing or asymmetric multiprocessing, or heterogeneous, with different core types. This influences substantially in the way applications share data.

Caches can be private to the cluster/tile, being coherent at that level, globally coherent in the chip, or not made coherent at all (*e.g.* [4] and [10]). As the number of cores increases, traditional solutions, such as buses or caches may become bottlenecks due to the contention on simultaneous accesses.

These challenging architectures introduce more complexity for sharing data between parallel threads. Lock-based synchronisation solutions are seldom used to avoid race conditions, but in multiprocessor systems, coarse-grained locks serialise non-conflicting operations that could progress in parallel, causing an impact on the system throughput, while fine-grained locks increase the complexity of system development, causing an impact on composability.

Alternatively, non-blocking approaches present strong conceptual advantages [19] and have been shown in several cases to perform better than lock-based ones [6]. The software transactional memory (STM) [18], is a concept in which a critical section – the *transaction* – executes in isolation, without blocking, regardless of other simultaneous transactions. An optimistic concurrency control mechanism is responsible to serialise concurrent transactions, maintaining the consistency of shared data objects. Conflicts are solved applying a contention policy that selects the transaction that will commit, while the contender will most likely abort and repeat. Solutions must be devised that reduce contention.

The time overhead resulting from aborts affects the worst-case execution time (WCET) of a task that executes a transaction. Therefore, the timing behaviour of a task can only be predictable if the transaction overhead is bounded, allowing to determine the WCET and the utilisation of the task. Additionally, minimising the number of aborts reduces wasted execution time.

In this paper, we formalise a FIFO-based contention management algorithm and two non-preemptive scheduling strategies that provide predictability and prevent transaction starvation. We evaluate the behaviour of these strategies by simulation, analysing the introduced overhead and consequent impact in schedulability.

The paper is structured as follows. Section 2 describes the problem of guaranteeing timing requirements when using STM in embedded real-time systems based on parallel architectures, and presents relevant published work in this field. Section 3 sets the system model in which the assumptions of this work are valid. We then formalise a decentralised algorithm to manage conflicts between concurrent transactions (Section 4). This contention management policy is more effective if transactions are not preempted, as we show in Section 5. The results from simulations that compare the performance of the contention management algorithm, under the two proposed scheduling strategies against pure partitioned EDF (P-EDF) are presented in section 6. This paper terminates with the conclusions and perspectives for further work in Section 7.

2 Background and Related Work

Transactional memory promises to ease concurrent programming: the programmer must indicate which code that forms the *transaction*, and relies an underly-

ing mechanism that maintains the consistency of shared data objects located at the *transactional memory*. Multiple transactions can be executed optimistically in parallel; however, when conflicting concurrent object accesses occurs (either a *read-write* or a *write-write* conflict) a contention policy is applied to guarantee the serialisation of the concurrent schedules, usually allowing one transaction to complete and aborting (and, consequently, repeat) the contenders. This approach has proved to scale well with multiprocessors [8], delivers higher throughput than coarse-grained locks and does not increase design complexity as fine-grained locks do [16].

STM achieves better performances when contention is low, causing low transaction abort ratio. Thus, STM behaves very well in systems exhibiting the following characteristics [11]: a predominance of read-only transactions, short-running transactions and a low ratio of context switching during the execution of a transaction. Some transactions may present characteristics (*e.g.* long-running, low priority) that can potentially expose them to starvation. In parallel systems literature, the main concern about STM is on system throughput, and the contention management policy has often the role to prevent livelock (a pair of transactions indefinitely aborting each other) and starvation (one transaction being constantly aborted by the contenders), so that each transaction will eventually conclude and the system will progress as a whole.

In real-time systems, the guarantee that a transaction will *eventually* conclude is not sufficient to assure the timing requirements that are critical to such type of systems: it must be known *how long* it will take to conclude. The verification of the schedulability of the task set requires that the WCET of each task is known, which can only be calculated if the maximum time used to commit the included transaction is known. As such, STM can be used in real-time systems as long as the employed contention management policy provides guarantees on the maximum number of retries each transaction is subject to.

Although the concept of STM is not new and numerous works have been published, 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 preempted by a higher-priority task is immediately aborted, and its effects undone. As no concurrent transactions are allowed in the system, it is impractical in multiprocessor systems. However, this policy matches with the *Abort-and-Restart* model [15].

In [2], and based on previous work on lock-free objects, Anderson *et al.* establish scheduling conditions for lock-free transactions under Earliest Deadline First (EDF) and Deadline Monotonic (DM), exclusively for uniprocessor systems. A different approach to support transactions in multiprocessor systems is provided in [1]: a wait-free mechanism relies on a helping scheme that provides an upper bound on the transaction execution time. In this approach, an arriving transaction must help pending transactions before being able to proceed, even

if no conflicts would occur, so the upper bound is likely to increase with the number of processors in the system.

In [9], Fahmy *et al.* describe an algorithm 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 for small atomic regions, assuming that any transaction will execute in, at most, two quanta.

Sarni *et al.* propose real-time scheduling of concurrent transactions for soft real-time systems in [17]. The authors adapted a practical STM to run on a real-time kernel, and modified the contention manager to apply their proposed policy. In this model, transactions are characterised by scheduling parameters, which are taken into account whenever solving a detected conflict between transactions. Conflicting transactions are serialised based exclusively on their absolute deadlines, which may have a negative effect on transactions with further deadlines.

In [3], we defend a FIFO-based approach to serialise concurrent transactions as a means to predict the time required to commit, but only provide a sketch of the decision algorithm. However, this paper does not take into account the considerable effect of preempting transaction on the predictability of the time required to commit.

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 useless processor utilisation, while assuring deadlines are met.

3 System model

We assume that jobs are released by a set of periodic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$, and scheduled on m identical processors denoted $P = \{P_1, \ldots, P_m\}$, under partitioned EDF (each task is statically assigned to a processor and each processor schedules its set of tasks under classical EDF). Each task τ_i is characterised by the period of job arrivals T_i, the worst-case execution time C_i, and the relative deadline D_i. The j^{th} job of task τ_i, hence forward denominated $\tau_{i,j}$, is characterised by the time the job arrives r_{ij}, and the absolute deadline of the job d_{ij}, defined as

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

For the sake of simplicity, we assume that each task τ_i performs at most one transaction, ω_i. Nevertheless, the results of this paper are extensible to tasks executing multiple non-nested transactions. Each transaction is characterised by:

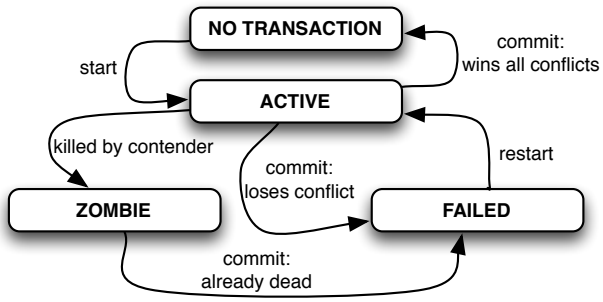

Fig. 1. State diagram of a transaction.

- C_{Ti} – the maximum execution time required to execute the sequential code once and try to commit,
- $DataSet_i$ – the data set, a collection of shared objects accessed by the transaction, which is divided into
- $ReadSet_i$ – the read set, the collection of objects that are accessed solely for reading, and
- $WriteSet_i$ – the write set, a collection of objects that are modified during the execution of the transaction.

A collection of STM objects $O = \{o_1, \ldots, o_p\}$ are assumed to be located at shared memory, being globally accessible to tasks, independently of the processor in which transactions are executing. We assume multiple simultaneous transactions are supported, and for each object there is a chronologically ordered list that contains records of all transactions currently accessing the object.

Each instance of a transaction has a life cycle that follows the states represented in Figure 1. Once a transaction arrives, it executes the transaction code and then tries to commit; if no conflicts are detected, the transaction commits, otherwise it may be aborted, retrying immediately. A transaction may be aborted multiple times until successfully commit. *Transaction overhead* is the execution time wasted executing aborted attempts. Transaction overhead of task τ_i is given considering the maximum number of failed attempts experienced by any of its jobs, $aborts_i$ before the transaction commits:

$$overhead_i = aborts_i \cdot C_{Ti}. \tag{2}$$

The WCET of a task that executes a transaction is then given by the time required to execute the code of the task without aborts C'_i, with the maximum transaction overhead:

$$C_i = C'_i + overhead_i. \tag{3}$$

The utilisation of this task is expressed by

$$U_i = C_i/T_i. \tag{4}$$

4 Contention management

A STM contention management policy oriented for real-time systems must tackle three issues.

- *Predictability.* When a transaction arrives, it must be assured that it will not exceed a determined time to commit (thus, imposing an upper bound on the number of aborts).
- *Starvation avoidance.* The ability to commit must be distributed fairly between contending transactions, so no task will have an excessive abort overhead.
- *Decentralised contention management.* The algorithm that implements the contention management policy should be preferably decentralised and executed by each transaction at the moment it tries to commit, and not on a dedicated processor.

In [3], we advanced that these issues are covered by a policy that sequences contending transactions by their chronological order of arrival, i.e. by the moment a transaction starts executing its first attempt. The algorithm we now formalise considers only static parameters, such as time of arrivals and core ids, so it is simple to reach a decentralised consensus. The time until commit should depend solely on the ongoing transactions at the moment the transaction starts, and independent of future arrivals of other transactions.

In algorithm 1 we detail the operations executed by every transaction when trying to commit. If the transaction was not previously turned in to *zombie*, it will take ownership of the objects in its data set. A transaction will have to wait that an object is released before taking ownership of it. For every object in its write set, the transaction determines if it wins against all contenders. In order to preserve work, the transaction just considers the contenders that are currently *active* and *running* (i.e. the host job is not preempted). If it fails on one object, then it immediately releases all owned objects, aborts and repeats. If all conflicts are won, it can immediately release the objects in the read set, commit updates, and mark the contenders in the write set as *zombies*, before releasing the objects.

Unlike locking solutions, the shared objects are just owned during the process of commit, and not during the whole critical section, which improves parallelism. Furthermore, the ownership process is controlled by the STM and should be transparent to the programmer, which improves composability.

However, if we consider preemptions during the execution of transactions, this behaviour can be seriously undermined, as we demonstrate in the next section.

5 Scheduling transactions

The way tasks are scheduled on multiple cores can affect the contention management behaviour and influence the success rate and predictability of transactions. Our algorithm permits that a transaction overtakes a preempted transaction:

Algorithm 1 STM contention management algorithm proposed for real-time systems.

Require: Current job of task τ_i has finished executing transaction ω_i.
Ensure: Transaction ω_i commits if and only if it wins all conflicts.

```
1: if ω_i status is ACTIVE then
2:    for all o_k ∈ DataSet_i do
3:       if ω_i status is ACTIVE then
4:          Take ownership of o_k
5:          for all ω_j contending with ω_i on o_k do
6:             if ω_j status is ACTIVE then
7:                if τ_j status is RUNNING then
8:                   if arrival(ω_i) > arrival(ω_j) then
9:                      Set ω_i status as FAILED
10:                  else if arrival(ω_i) = arrival(ω_j) and
                     Core(τ_i) > Core(τ_j) then
11:                     Set ω_i status as FAILED
12:      else
13:         Stop checking further objects
14:   if ω_i status is ACTIVE then
15:      for all o_k ∈ ReadSet_i do
16:         Remove ω_i entry from list
17:         Release o_k
18:      Commit updates
19:      for all o_k ∈ WriteSet_i do
20:         Remove ω_i entry from list
21:         for all ω_j accessing o_k do
22:            Set ω_j status as ZOMBIE
23:         Release o_k
24:   else
25:      Release all currently owned objects
26:      Abort and repeat ω_i
27: else
28:    Abort and repeat ω_i
```

this avoids deadlock between conflicting transactions executing in the same core, and preserves work of running transactions ready to commit. However, this reduces the probability of committing transactions that are prone to be preempted (e.g. long transaction, or low-priority job). Furthermore, a frequently preempted transaction may fail to commit for contending transactions that are more recent but are allowed to commit while the transaction is preempted, inverting artificially the intended behaviour of the system.

Cancelling temporarily preemptions, during the execution of a transaction, solves the problems of long transaction starvation and unpredictability stated above. If a transaction is guaranteed that it will not be preempted, then the success of transaction will depend solely on the contention management policy.

This solution can be compared with priority boosting [14]: raising the priority of a job to the highest level during a critical section cancels effectively preemptions. The Flexible Multiprocessor Locking Protocol (FMLP) [5] follows a similar approach to the one presented in this paper, executing critical sections non-preemptively, by their order of arrival. However, FMLP can serialise critical sections with non-intersecting datasets but accessing objects in the same group; STM allows such transactions to proceed and commit in parallel.

In this paper, we consider two non-preemptive approaches:

- *Non-preemptible until commit (NPUC).*
  In this approach, the job is assured to be scheduled from the moment the transaction arrives until it successfully commits.
- *Non-preemptible during attempt (NPDA).*
  In this approach, the task is non-preemptible during the transaction, but has preemption points between attempts. Any higher-priority job can be scheduled at any of these points.

5.1 Non-preemptible until commit

Under NPUC, each transaction will take-over the core in which it is executing, and will fail until all active direct contenders (transactions whose data accesses will conflict with the accesses of the transaction in consideration) that arrived earlier have committed and finished.

NPUC is totally predictable, as the time required for the transaction to successfully commit depends solely on the transactions that are already executing when the transaction arrives. Since direct contenders (transactions that have, at least, one conflict with the write set) will also wait for their own earlier direct contenders to finish, contention is propagated in chain. So, in the worst case, a transaction will have to wait for $(m-1)$ transactions to complete, assuming that every other core is already executing one transaction.

The predictability given by NPUC comes with a cost: higher priority tasks will have their responsiveness reduced due to lower-priority blocking.

5.2 Non-preemptible during attempt

Under NPDA, preemptions are limited during a transaction to preemption points inserted between attempts. This policy assures that the success of each attempt depends only on the running transactions, and reduces lower-priority blocking as compared with NPUC, improving responsiveness of higher-priority tasks.

Since jobs can be preempted between transaction attempts, one core can hold more than one ongoing transaction at any given time. This means that the number of earlier conflicting transactions for a given transaction is not limited to the number of remaining cores ($m - 1$), as in NPUC. So, although NPDA increases responsiveness of higher priority jobs, it is less predictable than NPUC.

6 Simulation results

We developed a simulation environment to test the proposed contention management algorithm under different scheduling policies – pure P-EDF, P-EDF with NPUC and P-EDF with NPDA – in systems containing from 2 to 64 cores.

For each system we generated randomly 20 synchronous task sets for three degrees of contention. The degree of contention is characterised by the ratio of the sum of all dataset sizes and the number of TM objects. In the experiments, we used 1.2, 2.4 and 3.6 ratios. All task sets demand each core a maximum ideal utilisation (not considering abort overhead) of 0.75.

In each task set, all tasks executed one transaction, and 50% were update transactions. The sequential execution time of each transaction C_T was 20% of the ideal WCET (without abort overhead) of the task. Each transaction could access, at most, 5 shared TM objects.

In each simulation, we recorded for each task the maximum number of aborts experienced in a job, the total number of aborts, the number of deadlines missed and the total execution time. For every task set, we simulated 10^6 time units under P-EDF, NPUC and NPDA.

First, we wanted to know how cancelling preemptions would affect the maximum number aborts experienced in a job of a task. We normalised the maximum number of aborts experienced in a job in NPUC and NPDA simulations to the values recorded in P-EDF simulations. Figure 2 presents the averages of these normalised results, indicating that cancelling preemptions tends to reduce the maximum number of aborts.

Next, we wanted to observe how the execution time would increase, due to aborts. Figure 3 shows the increase in execution time due to transaction overhead. We can observe that the amount of execution time with aborted transactions increases with contention density, as expected. The non-preemptive approaches also present very similar overheads (they overlap in this chart), and are lower than P-EDF, which means that they tend to produce less aborts, at the overall system perspective.

Table 1 reveals the total number of missed deadlines for each set of 20 simulations. Deadline misses are practically due to a very limited number of tasks

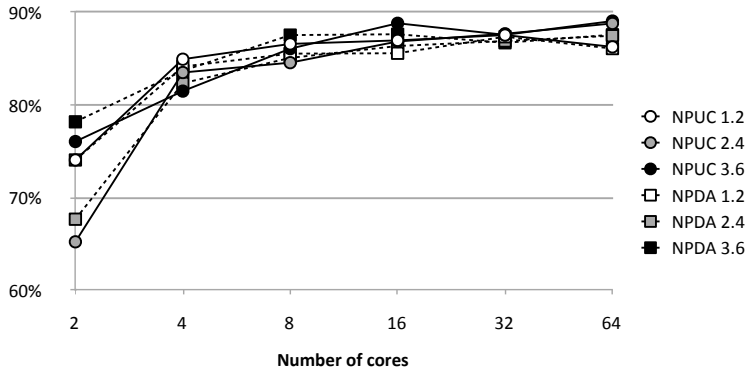

Fig. 2. Maximum aborts per transaction, normalised to P-EDF.

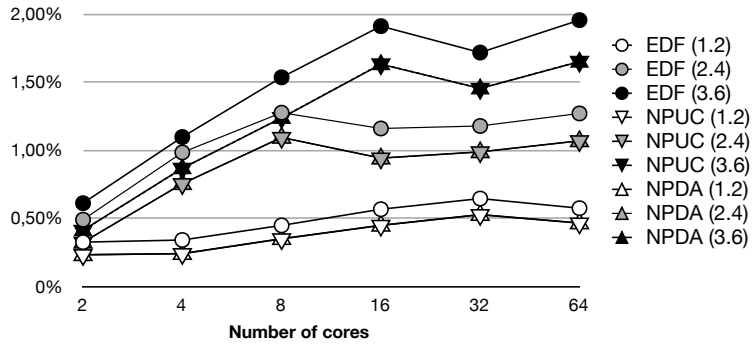

Fig. 3. System overall transaction overhead.

	1.2			2.4			3.6		
Cores	**EDF**	**NPUC**	**NPDA**	**EDF**	**NPUC**	**NPDA**	**EDF**	**NPUC**	**NPDA**
2	0	0	0	0	1	0	0	0	0
4	0	0	37	0	35	1	21	1	1
8	0	77	0	657	520	540	964	959	954
16	6	60	49	154	256	220	1939	1897	1687
32	33	204	147	483	723	589	1768	2317	1884
64	246	450	310	1400	1923	1424	4843	4757	4389

Table 1. Total deadlines missed (20 simulations).

with very high ideal utilisations, close to 0.75, and small periods. Inspection of simulation logs reveals that such tasks have laxities that are barely sufficient to accommodate transactions from concurrent tasks on the same core (in low contention scenarios), or accommodate multiple aborts (in higher contention scenarios). These characteristics do not fit non-preemptive approaches.

These results suggest that STM can naturally adapt to systems in which cores are grouped in tiles of 8 or 16 cores, and STM is isolated inside each partition (tile). Note that this maps well with cluster based many-core architectures which are emerging, where dozens or hundreds of processors are grouped into 8 or 16 core shared memory partitions, being the clusters interconnected by NoC.

7 Conclusions and further work

In this paper we propose a decentralised contention management algorithm for Software Transactional Memory (STM), for multi-core real-time systems, in which conflicting transactions are serialised by their chronological order of arrival. This algorithm is fair and avoids starvation across transactions. However, preempting a transaction reduces the probability of successfully commit, and so we propose two approaches to limit preemptions: non-preemptive until commit (NPUC) and non-preemptive during attempt (NPDA). NPUC is more predictable, while NPDA improves responsiveness of more urgent tasks.

Simulation results show that non-preemptive approaches can reduce transaction overhead. However, judicious processor allocation is required for tasks that have small laxity to accommodate transaction retries.

Acknowledgement. We would like to thank the anonymous reviewers for their suggestions and comments. This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects FCOMP-01-0124-FEDER-015006 (VIPCORE) and FCOMP-01-0124-FEDER-037281 (CISTER); by FCT and EU ARTEMIS JU, within project ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO).

References

1. Anderson, J.H., Jain, R., Ramamurthy, S.: Implementing hard real-time transactions on multiprocessors. In: Real-Time Database and Information Systems: Research Advances, pp. 247–260. Kluwer Academic Publishers, Norwell, MA, USA (Sep 1997)
2. Anderson, J.H., Ramamurthy, S., Moir, M., Jeffay, K.: Lock-free transactions for real-time systems. In: Real-Time Database Systems: Issues and Applications, pp. 215–234. Kluwer Academic Publishers, Norwell, MA, USA (May 1997)
3. Barros, A., Pinho, L.M.: Software transactional memory as a building block for parallel embedded real-time systems. In: Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011). Oulu, Finland (Aug 2011)
4. Benini, L., Flamand, E., Fuin, D., Melpignano, D.: P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In: Proceedings of theConference & Exhibition Design, Automation Test in Europe (DATE 2012). pp. 983 –987 (Mar 2012)

5. Block, A., Leontyev, H., Brandenburg, B.B., Anderson, J.H.: A Flexible Real-Time Locking Protocol for Multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007). pp. 47–56. Daegu, South Korea (Aug 2007)
6. Brandenburg, B.B., Calandrino, J.M., Block, A., Leontyev, H., Anderson, J.H.: Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08). pp. 342–353 (Apr 2008)
7. Choi, B., Komuravelli, R., Sung, H., Smolinski, R., Honarmand, N., Adve, S., Adve, V., Carter, N., Chou, C.T.: Denovo: Rethinking the memory hierarchy for disciplined parallelism. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 155 –166 (Oct 2011)
8. Dragojević, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Communications of the ACM 54(4), 70–77 (Apr 2011)
9. Fahmy, S.F., Ravindran, B., Jensen, E.D.: On Bounding Response Times under Software Transactional Memory in Distributed Multiprocessor Real-Time Systems. In: Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE '09). pp. 688–693 (Apr 2009)
10. Kalray: MPPA 256 – Many-core processors (2012), `http://www.kalray.eu/products/mppa-manycore/mppa-256/`
11. Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J.L., Muller, G.: Scheduling support for transactional memory contention management. In: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10). pp. 79–90 (Jan 2010)
12. Manson, J., Baker, J., Cunei, A., Jagannathan, S., Prochazka, M., Xin, B., Vitek, J.: Preemptible Atomic Regions for Real-Time Java. In: Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05). pp. 62–71. Miami, FL (Dec 2005)
13. Martin, M.M.K., Hill, M.D., Sorin, D.J.: Why on-chip cache coherence is here to stay. Communications of the ACM 55(7), 78–89 (Jul 2012)
14. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the 10th International Conference on Distributed Computing Systems. pp. 116–123 (1990)
15. Ras, J., Cheng, A.M.K.: Response time analysis for the Abort-and-Restart event handlers of the Priority-Based Functional Reactive Programming (P-FRP) paradigm. In: Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). pp. 305–314 (2009)
16. Rossbach, C.J., Hofmann, O.S., Witchel, E.: Is transactional programming actually easier? In: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10). pp. 47–56 (Jan 2010)
17. Sarni, T., Queudet, A., Valduriez, P.: Real-Time Support for Software Transactional Memory. In: Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 2009). pp. 477–485 (Aug 2009)
18. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC '95). pp. 204–213 (Aug 1995)
19. Tsigas, P., Zhang, Y.: Non-blocking data sharing in multiprocessor real-time systems. In: Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA'99). pp. 247–254 (Dec 1999)