Technical Report

Software transactional memory as a building block for parallel embedded real-time systems

António Barros

Luís Miguel Pinho

HURRAY-TR-110706

Version:

Date: 07-25-2011

Software transactional memory as a building block for parallel embedded 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 recent trends of chip architectures with higher number of heterogeneous cores, and non-uniform memory/non-coherent caches, brings renewed attention to the use of Software Transactional Memory (STM) as a fundamental building block for developing parallel applications. Nevertheless, although STM promises to ease concurrent and parallel software development, it relies on the possibility of aborting conflicting transactions to maintain data consistency, which impacts on the responsiveness and timing guarantees required by embedded 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 assess the use of STM in the development of embedded real-time software, defending that the amount of contention can be reduced if read-only transactions access recent consistent data snapshots, progressing in a wait-free manner. We show how the required number of versions of a shared object can be calculated for a set of tasks. We also outline an algorithm to manage conflicts between update transactions that prevents starvation.

Software transactional memory as a building block for parallel embedded real-time systems

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

***Abstract*—The recent trends of chip architectures with higher number of heterogeneous cores, and non-uniform memory/non-coherent caches, brings renewed attention to the use of Software Transactional Memory (STM) as a fundamental building block for developing parallel applications. Nevertheless, although STM promises to ease concurrent and parallel software development, it relies on the possibility of aborting conflicting transactions to maintain data consistency, which impacts on the responsiveness and timing guarantees required by embedded 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 assess the use of STM in the development of embedded real-time software, defending that the amount of contention can be reduced if read-only transactions access recent consistent data snapshots, progressing in a wait-free manner. We show how the required number of versions of a shared object can be calculated for a set of tasks. We also outline an algorithm to manage conflicts between update transactions that prevents starvation.**

I. INTRODUCTION

The current trend to increase processing power by manufacturing chips including multiple processor cores has popularised 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[1], as software performance must rely on efficient techniques to design and execute concurrent software in parallel.

The real-time systems community has established strong scheduling and synchronisation theories and techniques that are essential to verify and guarantee the timing requirements of any set of concurrent tasks, executing on a uniprocessor system. Avoiding race conditions with lock-based synchronisation became commonplace, despite the well-known pitfalls: complexity, lack of composability [2] and (bounded) priority inversion. In multiprocessor systems, lock-based synchronisation introduces additional drawbacks. Coarse-grained locks serialise non-conflicting operations that could progress in parallel, and may cause cascading or convoying blocks [3], impairing concurrency with an impact on system throughput. Fine-grained locks increase the complexity of system design, hindering composability.

On multiprocessors, non-blocking approaches present strong conceptual advantages [4] and may perform better than lock-based ones [5]. One concept under research is the *software transactional memory* (STM) [6], in which a critical section – the *transaction* – executes isolated from other simultaneous transactions, without blocking. An optimistic concurrency control mechanism preserves the consistency of shared data, generally aborting selected transactions to solve data access conflicts.

The number of times a transaction is aborted reflects on the execution time of the host job. Therefore, this value must be limited in order to compute the worst-case execution time (WCET) of the job, and minimised to reduce the processor capacity used with wasted work. In this paper, we defend two approaches to reduce and limit the amount of transaction aborts suffered by every transaction:

- **Using multi-versioned STM.** Read-only transactions can be serialisable if executed with a recent and consistent snapshot of their read-sets without conflicting with other concurrent transactions, as long as the STM keeps multiple versions of each object, reducing the overall frequency of transaction aborts. We demonstrate how the timing characteristics of real-time systems can be used to determine the maximum number of versions for each data object (Section II).
- **Preventing starvation using scheduling data.** Conflicts are unavoidable, and transaction starvation increases abnormally the execution time of a task. We outline an algorithm that prevents indefinite aborts (Section V), setting a limit on the number of aborts for each transaction.

The paper is structured as follows. Section II 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 III sets the system model in which the assumptions of this work is valid. We show how to determine the exact number of versions each shared object must store on a

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

real-time system using multi-version STM in Section IV. Additionally, we outline a distributed algorithm to manage conflicts between concurrent transactions that limits the number of aborts for each transaction (Section V). This paper terminates with the conclusions and perspectives for further work in Section VI.

II. Background and Related Work

Transactional Memory promises to ease concurrent programming: the programmer marks the transactional code and leaves the burden of synchronisation details to an underlying mechanism that must maintain the consistency of shared data located at the *transactional memory*. Multiple transactions can be executed optimistically in parallel; when conflicting accesses to an object occur, a contention policy serialises the concurrent schedules, generally selecting one transaction to commit and aborting-and-repeating the contenders. This approach scales well with multiprocessors [7], delivers higher throughput than coarse-grained locks and does not increase design complexity as fine-grained locks do [8].

STM achieves better performances when contention is low, specially when exists a predominance of read-only transactions, short-running transactions and a low ratio of context switching during the execution of a transaction [9]. Reducing the probability of conflicts may have a beneficial effect on the schedulability of the task set. Removing read-only transactions from the conflict arena, using multi-versioned STM is one option to that end.

Still, under contention, some transactions may present characteristics (*e.g.* long running, low priority) that can potentially expose them to starvation. 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), so that every transaction will eventually commit.

The guarantee that a transaction will *eventually* commit does not assure the critical timing requirements of real-time systems: it must be known *how long* it will take to commit. The schedulability analysis of the task set requires that the WCET of each task is known, and that includes the maximum time required for the transaction to commit. As such, STM can only be used in real-time systems if the contention management policy provides guarantees on the maximum number of retries for each transaction.

Although the concept of STM is not new and numerous works have been published, only some few dealt with it in the context of real-time systems. Early work covered transaction support in uniprocessor real-time systems, with Anderson *et al.* [11] establishing scheduling conditions for lock-free transactions under EDF and DM. More recently, Manson *et al.* [10] presented an analysis to bound the response time of jobs with atomic regions, but the system model does not allow concurrent transactions.

Considering transaction support in multiprocessor platforms, Anderson *et al.* [12] described a wait-free mechanism that provides an upper bound on the transaction execution time, but the helping scheme employed is pessimistic and increases the upper bound with the number of processors. Fahmy *et al.* [13], described how to calculate an upper-bound on the WCET of tasks containing multiple transactions, scheduled under Pfair; however, transactions must be limited in duration. Sarni *et al.* [14], adapted a practical STM to a real-time kernel, and modified the contention manager to decide based on the absolute deadlines associated with each transaction (EDF); this approach can increase the abort ratio of transactions with further deadlines, and allows deadlines to be missed. Finally, Schoeberl *et al.* [15] demonstrated that a task containing a single transaction, executing on a system with hardware TM, will meet deadlines as long as the transactions of two consecutive jobs are separated by the *resolve time*, but conflicts are not solved based on on-line scheduling data.

Concerning multi-versioned STM, published works either store an arbitrary fixed number of versions for each object (just reducing the probability of read-only aborts) such as in [16] or a variable number of versions dynamically managed by a garbage collector, which eliminates read-only transaction aborts [17]–[19] but does not suit the timing requirements of real-time systems.

These works provide already some perspectives on how to deal with STM in real-time systems, but there are many issues pending, so 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. System model

The system model assumes that jobs are released by a set of periodic tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ with implicit deadlines, and scheduled on m identical processors denoted $P = \{P_1, \ldots, P_m\}$, under partitioned EDF. Each task τ_i is characterised by (T_i, C_i), being T_i the period of job arrivals and C_i the worst-case execution time. 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

$$d_{ij} = r_{ij} + T_i. \quad (1)$$

In this analysis, we assume each task τ_i performs at most one transaction, TRX_i, characterised by:

- W_i – the maximum execution time necessary to execute once the sequential code,
- RS_i – the read-set, the collection of objects that are accessed solely for reading, and

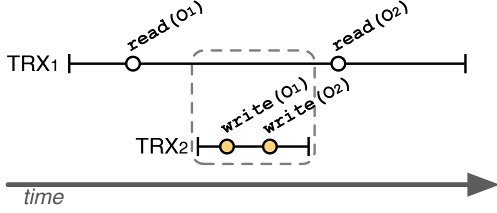

Figure 1. A read-only (TRX_1) and an update (TRX_2) transactions contend on access to objects O_1 and O_2.

- WS_i – the write-set, the collection of objects that are modified during the execution of the transaction.

When 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, restarting immediately. A transaction may be aborted multiple times until successfully commits. *Transaction overhead* is the additional execution time required by the n_{ij}^{aborts} aborted retries before the transaction commit, and is given by

$$W_{ij}^{overhead} = n_{ij}^{aborts} \times W_i. \quad (2)$$

A transaction can abort while trying to commit, if it finds conflicting data accesses and the contention policy dictates that another contending transaction must conclude before it.

A collection of STM objects $O = \{O_1, \ldots, O_p\}$ are assumed to be globally accessible to tasks, independently of the processor in which transactions are executing, and multiple simultaneous transactions are supported.

The response time of a job is here defined as the time elapsed since a job arrives until its execution is completed. 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 if executed sequentially (without aborts) C_i', the interference time I_{ij} in which the job was pre-empted by higher-priority tasks and the overhead due to the n_{ij}^{aborts} aborted executions of the transaction. The response time of a job can be defined by

$$RT_{ij} = C_i' + I_{ij} + n_{ij}^{aborts} \times W_i \quad (3)$$

and to meet deadlines, the response time must be

$$RT_{ij} \leq T_i \quad \forall i, j \quad (4)$$

IV. Reducing contention using multi-versioned STM

Previous work on STM has consistently shown that the amount of contention has a relevant impact on the behaviour of a STM. Under this evidence, reducing contention will expectedly improve the performance of the system.

A conflict occurs when concurrent transactions access the same data location and at least one of the transactions updates it. Considering the example in Figure 1, transaction TRX_1 reads object O_1 and, later on, reads object O_2; in between these two accesses, transaction TRX_2 modifies both objects and tries to commit. This concurrent execution is not serialisable because it does not produce the same outcome as if TRX_1 was executed sequentially before or after TRX_2. In this situation, one of the transactions must abort so the contender can commit.

With multi-versioned STM, read-only transactions work on recent consistent snapshots of their read-sets without ever conflicting with other concurrent transactions [17] and, therefore, commit at first try. Considering the same example in Figure 1, both transactions can now commit, as long as TRX_1 has access to the version of O_2 previous to the update performed by TRX_2: although TRX_1 commits after TRX_2, it can be serialisable as if it has executed before TRX_2.

The amount of contention on object accesses is reduced at the expense of higher memory utilization to temporarily store previous version of each object. The amount of memory overhead that optimises the system throughput is a subject of current research in the field of parallel systems [16], [18]–[20], essentially relying on fixed number of versions or in garbage collecting techniques that will statistically reduce or eliminate the abort ratio of read-only transactions.

In real-time systems, the timing characteristics of the task set and the data set of each transaction are known beforehand, allowing to determine the exact number of versions required for every object. Knowing the exact number of versions each object must store, permits to design a STM with minimum memory overhead without garbage collecting mechanisms, and guaranteeing read-only transactions will never conflict with concurrent transactions.

To determine the number of versions required for any object we have to:

1) determine the maximum number of updates for each object in a given interval, and
2) determine the time each object must store a version.

For an arbitrary time interval ΔT we can calculate the maximum number of updates of an object O_k – denoted as $N_k^{updates}$ – considering the timing properties of the tasks that modify the object, given by the number of job releases of tasks that host transactions including O_k in its write-set:

$$N_k^{updates} = \sum_i a_i \times \left\lceil \frac{\Delta T}{T_i} \right\rceil \quad (5)$$

in which a_i is given by

$$a_i = \begin{cases} 1 & \text{if } O_k \in WS_i, \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Each version of object O_k must be stored for the maximum time a read-only transaction including the O_k in its read-set can execute. Pessimistically, we assume the object may be read any time during the period of the task. Thus, a version of object O_k must be stored for

$$T_k^{store} = \max\{T_i : O_k \in RS_i \wedge TRX_i \text{ is read-only}\}. \quad (7)$$

Combining the two results from Equations (5) and (7), the number of versions required for O_k is given by

$$N_k^{versions} = \sum_i a_i \times \left\lceil \frac{T_k^{store}}{T_i} \right\rceil \quad (8)$$

in which a_i is given by

$$a_i = \begin{cases} 1 & \text{if } O_k \in WS_i, \\ 0 & \text{otherwise.} \end{cases} \quad (9)$$

Therefore, multi-versioned STM can be implemented efficiently in real-time systems with predetermined memory overhead, and assuring all read-only transactions will execute in a wait-free manner.

V. CONTENTION MANAGEMENT

Conflicting concurrent object accesses must be solved by a contention management policy that is responsible to maintain object consistency, serialising contending transactions according to a criteria that represents the expected behaviour of the system. A contention management policy must avoid live-lock situations, in which a group of transactions are indefinitely aborted without ever committing, and must prevent transactions from starving because of some of their inherent characteristics.

In this paper we outline an approach that tackles three important issues.

- **Predictability.** Predictability is one of the most important requirements in real-time systems. When a transaction arrives, we want to be assured that it will not exceed a determined time to commit, thus we need a limit on the number of aborts. Full control on managing conflicts with update transactions implies *read operations must be visible*, allowing to abort a transaction that is trying to modify an object in favour of a contender that is merely reading the same object.
- **Liveliness.** We want to ensure liveliness will be distributed fairly between contending transactions. If a transaction is overlooked by the contention management policy and gets excessively aborted, then the host job will have its execution time increased and may end up monopolising the processor in which it is executing, disturbing the execution of the local task set. Refraining excessive aborts on each transaction directly hinders this type of abnormal behaviour.
- **Distributed contention management.** The algorithm must be distributed and executed by each transaction at the moment it tries to commit. In case of a conflict, all involved transactions must reach a consensus on the transaction that will commit. The parallel nature of this algorithm avoids any possible bottlenecking problem that a centralised contention manager could present in a many-cores architecture.

Our approach sequences contending transactions by their chronological order of arrival, *i.e.* by the moment a transaction starts executing its first try. This criterion is fair in the sense that no transaction will be chronically discriminated due to some innate characteristic. The time overhead of a transaction depends solely on the ongoing transactions at the moment the transaction arrives, being independent of future arrivals of other transactions. It has the side-effect of a transaction not being able to commit before an older direct contender that is, in turn, waiting for a third transaction (that does not directly conflicts with the first one) to commit.

However, a transaction can overtake an older transaction that is, at that moment, pre-empted. This is absolutely necessary to avoid deadlock between conflicting transactions assigned to the same processor (*i.e.* a more recent transaction pre-empts an older one, but becomes unable to commit before the pre-empted transaction and enters in a try-abort infinite cycle) and prevents the transaction overhead be increased by the interference suffered by other concurrent jobs executing in other processors.

The probability of ties on the times of arrival is low but, if necessary, they are broken by two additional levels of decision: first by comparing the laxities of the jobs when the transactions arrived selecting the transaction with smaller laxity to succeed and second (if still necessary) selecting the transaction executing on the processor with lower identification to succeed.

When a transaction finishes to execute its sequential code and tries to commit, it checks for conflicts on every transactional object accessed. For every conflict detected, the transaction applies the algorithm to determine if it is has the means to commit. The transaction will finally commit if has won on all detected conflicts, otherwise it aborts and will have to repeat the transaction code, again.

VI. CONCLUSIONS AND FURTHER WORK

In this paper we assess the use of Software Transactional Memory (STM) as a building block for the development of parallel embedded real-time systems. We further propose the use of multi-version STM to reduce contention between transactions and to execute jobs with read-only transactions in a wait-free manner. Profiting the known timing characteristics of real-time task sets, we are able to calculate the exact number of versions each object must keep.

In this model, contention is only possible between update transactions. Conflicting transactions are serialised by their chronological order of arrival, applying an algorithm that is executed in parallel by each transaction when trying to commit. This approach avoids transactions to starve due to some inherent characteristic overlooked by the contention management policy. However, an effect of this model is that a transaction may have to wait for a direct contender to

commit that, in turn, is waiting for another transaction to commit, that is not conflicting with the first one.

The scenario considered on this paper is purposely pessimistic, because we do not want to make any assumptions on particular optimisations available on a practical STM. However, this work points directions for improvements that can reduce the time overheads of transactions, reducing wasted processor utilization and increasing throughput.

An immediate improvement could be achieved if a committing transaction could signal its contenders so they could restart immediately instead of letting them misspend execution time that will helplessly result in an abort. Another alternative would be for a committing transaction to mark their contenders as zombies, which would enable other transactions waiting on these zombie transactions to commit, as long as no other conflicts with active and sound transactions occurred.

Acknowledgment

This work was supported by FCT through the VipCore (PTDC/EIA-CCO/111799/2009) project, 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.

[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 *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS 1993)*, May 1993, pp. 264–273.

[4] P. Tsigas and Y. Zhang, "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)*, 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 *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, Apr. 2008, pp. 342–353.

[6] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC '95)*, Aug. 1995, pp. 204–213.

[7] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui, "Why STM can be more than a Research Toy," *Communications of the ACM*, Apr. 2011.

[8] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)*, Jan. 2010, pp. 47–56.

[9] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)*, Jan. 2010, pp. 79–90.

[10] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek, "Preemptible Atomic Regions for Real-Time Java," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, Dec. 2005, pp. 62–71.

[11] 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*. Norwell, MA, USA: Kluwer Academic Publishers, May 1997, pp. 215–234.

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

[13] S. F. Fahmy, B. Ravindran, and E. D. Jensen, "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)*, Apr. 2009, pp. 688–693.

[14] T. Sarni, A. Queudet, and P. Valduriez, "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)*, Aug. 2009, pp. 477–485.

[15] M. Schoeberl, F. Brandner, and J. Vitek, "RTTM: Real-Time Transactional Memory," in *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, Mar. 2010, pp. 326–333.

[16] T. Riegel, P. Felber, and C. Fetzer, "A Lazy Snapshot Algorithm with Eager Validation," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006, pp. 284–298.

[17] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Science of Computer Programming*, vol. 63, no. 2, pp. 172–185, Dec. 2006.

[18] D. Perelman, R. Fan, and I. Keidar, "On Maintaining Multiple Versions in STM," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC '10)*, Jul. 2010, pp. 16–25.

[19] D. Perelman and I. Keidar, "SMV: Selective Multi-Versioning STM," in *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Apr. 2010.

[20] T. Riegel and P. Felber, "Snapshot Isolation for Software Transactional Memory," in *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06)*, Jun. 2006.