Improving the Efficiency of a Hardware Transactional Memory on an NoC-based MPSoC

Leonardo Kunz, Gustavo Girão, Flávio R. Wagner
Institute of Informatics
Federal University of Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
{lkunz, ggbsilva, flavio}@inf.ufrgs.br

***Abstract*—Transactional Memories (TM) have attracted much interest as an alternative to lock-based synchronization in shared-memory multiprocessors. Considering the use of TM on an embedded, NoC-based MPSoC, this work evaluates a LogTM implementation. It is shown that the time an aborted transaction waits before restarting its execution (the backoff delay) can seriously affect the overall performance and energy consumption of the system. This work also shows the difficulty to find a general and optimal solution to set this time and analyzes three backoff policies to handle it. A new solution to this issue is presented based on a handshake between transactions. Results suggest up to 20% in performance gains and up to 53% in energy savings when comparing our new solution to the best backoff delay alternative found in our experiments.**

Keywords: Hardware Transactional Memories; Multiprocessor Systems-on-Chip; Networks-on-Chip; Embedded Systems; Performance; Energy Consumption.

I. INTRODUCTION

Transactional Memories (TM) [1] have been proposed as a solution for shared memory synchronization on multiprocessors as a more efficient alternative to the traditional lock mechanism. LogTM [2] has been one the most accepted hardware transactional memory (HTM) models.

A critical question that arises from the LogTM model is how much time an aborted transaction should wait before restarting its execution (hereafter named *Backoff Delay on Abort*). An underestimated time can cause many other aborts and even compromise system progress, while an overestimated time harms performance.

This work shows that the ideal value for the *Backoff Delay on Abort* is not easily predictable. We evaluate three different heuristic approaches (*backoff policies*) to handle this issue and present an alternative based on the use of a simple handshake mechanism to solve it. This new approach is called *Abort Handshake*. Experimental results suggest that performance gains reach up to 20% and energy savings reach up to 53% when the *Abort Handshake* mechanism is compared to the best backoff policy for each system configuration.

The major contributions of this work are to point out and analyze the problems with the *Backoff Delay on Abort* issue on LogTM, to propose a modification to solve them, and to study the potential of this modification to improve performance and energy consumption.

978-3-9810801-7-9/DATE11/©2011 EDAA

This paper extends our previous work [3], which evaluates energy consumption and performance of a LogTM implementation on a NoC-based embedded MPSoC and also presents a more detailed description of our LogTM implementation. The system's environment, as presented in Fig. 1, is composed by a configurable number of processors with private data caches, a shared memory and a directory module in order to provide cache coherence.

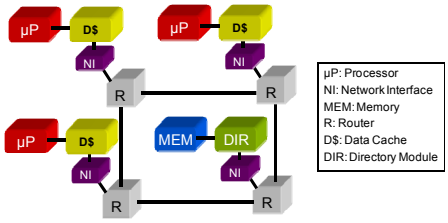

Figure 1. System's environment

The remaining of the paper is organized as follows. Section II details the backoff delay problem, and Section III presents the solution proposed in this work. Section IV discusses related work, while experimental results regarding the use of the *Abort Handshake* mechanism are presented in Section V. Section VI draws main conclusions and introduces future work.

II. BACKOFF DELAY ON ABORT ON LOGTM

LogTM and its variants have eager conflict detection, eager version management and requester stalls with conservative deadlock avoidance. Such systems stall the requester if a conflict is detected, but, if a potential deadlock cycle is detected, the requester aborts. Timestamps assigned to every transaction in the beginning of their execution are used to set priority – earlier transactions have higher priority when deciding which one should abort. If a later transaction stalls an earlier transaction by sending a *nack*, a per-processor flag called *possible cycle* is set. The later transaction aborts when it receives a *nack* from an earlier transaction, and its *possible cycle* flag is set. This mechanism avoids that a transaction waits for a later transaction while at same time it is forcing a later transaction to wait, thus eliminating potential deadlock cycles.

To enable conflict detection, two bits per block in each cache keep track of the blocks that have been read and written (R and W bits, respectively) during a transaction. They represent, roughly speaking, an "ownership" by the current transaction that the processor releases on an abort by clearing

them. The transaction that caused the other one to abort needs to retry its request in order to get the ownership of the conflicting block and continue execution. After abort, the processor waits for a backoff delay before restarting. If the aborted transaction restarts and retrieves the ownership of the conflicting block before the aborter[1], the same conflict will happen, causing another abort of the transaction. This situation may not only harm performance but also lead to livelocks if it repeats indefinitely.

Fig. 2 (left) shows an example where two transactions (T0 and T1, T0 with an earlier timestamp) have block A shared for read in both caches, and both try to write on this block. (1) As the block is already in its cache, T0 requests permission to write (GETP) to the directory (DIR), which forwards its request by asking T1 to invalidate its copy. T1 detects the conflict, sends a *nack* to T0 and sets its *possible cycle* flag, since T1 has a later timestamp than T0. On receiving the *nack*, T0 stalls and retries its request in (3). The second request the directory receives (2) is GETP from T1, incurring in T1 having to abort when it receives *nack* from T0. In this example, T1 requests block A for read and has a cache hit before the invalidation on (3) arrives to T1, thus causing the same conflict again.

We highlighted in Fig 2 two times: *retrieve time*, measured from the moment T1 releases ownership of the blocks on abort, until it reads or writes the block and gets the ownership again before T0; and the *ownership change time*, the time that the higher priority transaction takes to get the ownership of the block after the lower priority transaction releases it. Clearly, *retrieve time* must be longer than the *ownership change time* in order that the higher priority transaction acquires permission on the block.

Considering that the aborted transaction releases ownership after completing rollback and restoring memory, the *retrieve time* depends on two main components: 1) *Backoff Delay on Abort* – an arbitrary time the processor waits before restarting the aborted transaction; and 2) time for the transaction to access the conflicting block, including a possible re-fetch of the block if it gets invalidated or is replaced in cache. The *ownership change time* is hard to predict since it depends on the processing of coherence requests, including network delays. Besides the request coming from the aborter transaction, another request(s) from other processors in the system can be, misfortunately, processed earlier in the directory.

The original LogTM proposal handles the backoff issue with a randomized linear backoff after abort and optionally allows the use of more accurate (and complex) contention managers in software if the problem persists. This backoff strategy lets the system "adapt" to guarantee progress, but performance still remains a matter of tuning parameters. If the time is too short, the higher priority transaction may have to retry several times while the aborted processors may restart and abort again, executing useless work. With a time longer than needed, system can lose opportunity to explore more parallelism and the delay can impact the overall performance. The use of a contention manager in software can add significant overhead to the system, and, since there is no contention manager that performs better for all workloads, choosing one can be a complex task [4].

[1] On LogTM, a transaction does not abort directly another one, it aborts itself on receiving NACK. For clarification, we use the term "aborter" to specify the higher priority transaction that answered the NACK that caused the other transaction to abort.

III. Proposed Solution – The Abort Handshake

Instead of trying to find a better heuristic to set the *backoff delay after abort*, our approach is based on signaling between transactions in a way that the aborted transaction is notified when it is safe to restart. Two messages are used for this purpose. *Signal abort* is sent by the transaction as soon as it aborts to notify the processor that caused it to abort. *Release abort* is the answer in the opposite way to notify when the aborted transaction can restart execution. Each processor keeps a list of transactions that sent *Signal abort* to it during the current transaction and sends *Release aborts* to all processors in the list on commit.

Fig. 2 shows on the right the same case as in its left but now using our approach. When T1 aborts after conflicting with T0, it sends a *Signal Abort* to T0 and waits until it receives a *Release Abort*. T0, then, adds T1 to its list of transactions it has aborted. When it commits, T0 sends a *Release Abort* to T1 allowing it to restart.

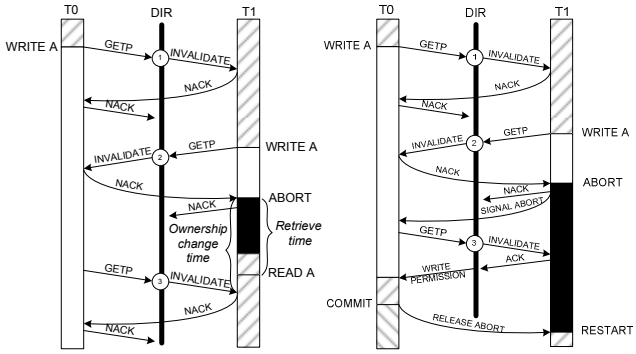

Figure 2. Successfull invalidation of block after abort (left) and the *abort handshake* solution for the same case (right)

In the case of multiple sharers of a block, a transaction may receive more than one *nack* for a single request. If it aborts, it chooses one of the higher priority transactions that sent *nack* to send its *Signal Abort*. A transaction can only wait for a higher priority transaction, avoiding the risk of deadlock. Also, a transaction that aborted another one can be aborted by a third transaction; one will notify the other after its commit, and so on.

This solution serializes the entire execution of two conflicting transactions when one of them aborts by allowing it to restart only after the commit of the other. It is a conservative approach as the aborted transaction may have work to execute independently before requesting the conflicted block again. The aborted transaction may not even request the conflicting block again if the decision to request it depends on data altered by a third transaction in the meantime. Nevertheless, this approach can save energy by reducing speculation, consequently reducing contention on the directory and on the interconnection network, and this can also yield performance gains if the speculation would not pay off.

IV. RELATED WORK

To the best of our knowledge, this work is the first one to discuss the *backoff delay on abort* issue in LogTM and to propose a solution that is not based on heuristic approaches.

Serialization of transactions after a conflict has been proposed by Moreshet et al. [5] to save energy and increase performance under high contention. However, the evaluation of their serialization mechanism is not conclusive enough since it uses only a synthetic microbenchmark and a single system configuration. Sanyal et al. [6] developed a scheme for clock gating processors on abort to save energy combined with a new contention manager policy for Scalable-TCC HTM. Ferri et al. [7,8] analyze energy consumption of TM for an embedded architecture using ARM7 processors connected by a bus. They propose many architectural modifications to reduce energy consumption on their implementation, which is based on the original HTM of Herlihy et al. [1]. All works reported above use buses for interconnection.

Our previous work [3] evaluates energy and performance of TM comparing to locks with a LogTM-based implementation on an NoC-based embedded MPSoC. As far as we know, it was the first one to implement TM over an NoC and the first one to address the energy consumption of LogTM.

V. EXPERIMENTS

A. Setup

The evaluation presented in this paper was performed in a virtual platform known as SIMPLE (Simple Multiprocessor Platform Environment) [9]. SIMPLE is a SystemC cycle-accurate virtual platform that emulates an NoC-based MPSoC. It is composed by Java Processors [10] and different memory organizations to be defined at design time. We used a shared memory organization, with a single memory node located in a way to minimize the average distance in hops from processors to memory on each configuration of a network-on-chip [11] with mesh topology. Each processor has an 8 KB fully-associative L1 cache, with a block size of 32B, and the main memory node has 64 KB.

The experiments evaluate performance, measured by the total execution time of each application, and the overall energy spent, including processors, network, and memories. For the energy of the processors, a cycle-accurate power simulator [12] is used. For the network (including buffers, arbiter, crossbar, and links), the Orion library [13] is applied, and for the memory and caches we use the Cacti tool [14]. All results were obtained for 180 nm technology.

For the evaluation, an adaptation of the LeeTM [15] benchmark is used. LeeTM is an implementation of the Lee's circuit routing algorithm, which is a realistic application with inherent parallelism, but hard to express with locks. Due to limitations of our platform, a simplified version of the input data is used. The input data contain 32 tracks to be routed, randomly distributed on a two-layer grid board represented by a 2x20x20 matrix of interconnection points.

To evaluate the behavior of the system in the case where there is no parallelism between the transactions, we artificially forced contention. To do that, we modified the application by adding a write in a shared variable at the beginning of each transaction. We force all transactions to retain exclusive ownership of this shared variable block during the entire transaction, thus inhibiting parallelism, but keeping the same input data for the application. This modified version of the benchmark, called LeeTM-IP, allow us to analyze the overhead of performance and energy of each synchronization mechanism when the amount of parallelism that TM can explore is the same as the lock version of the application.

B. Results

We compare our approach to three different backoff policies to handle the *backoff delay on abort*. Each policy has an input parameter that represents an initial reference for the number of cycles the transaction should wait before restarting. In order to explore a wide value range of the input parameter, we changed it from 625 to 640000 cycles with exponentially spaced intervals, doubling each time. The policies are the following:

Fixed Backoff – Transaction waits after abort for a constant delay defined by the input parameter before restarting.

Exponential Backoff – Initially the delay is specified by the input parameter, doubling it at every abort of the transaction. Commit restores the parameter to its initial value.

Randomized Linear Backoff – The delay is a random value chosen in each abort in the interval between zero and a value initially equal to the input parameter. On every abort the parameter is added to the interval, increasing it linearly. On commit, the interval returns to the initial value.

We firstly analyze the difficulty to set an appropriate parameter value to obtain the best performance for different system configurations. Fig. 3 shows the parameter values that give the best execution times for each combination of policy and number of processors in the system running the LeeTM benchmark. For each case, a different parameter value gives optimal performance. Furthermore, no policy presented a predictable behavior as the number of cores increases.

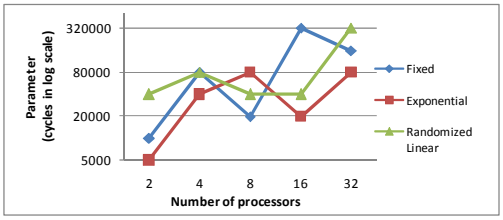

Figure 3. Optimal parameters of each backoff policy for LeeTM

Our second evaluation considers the impact of non-optimal parameters on the performance of the application. We selected the optimal values for eight processors (8P) in each policy and executed the application with different numbers of processors and keeping the same parameters. Fig. 4 shows the relative difference of cycles normalized to the optimal execution time among the three backoff policies for the corresponding number of processors. The execution on 32P with fixed policy using 20000 as input parameter livelocked. In an extreme case, the execution time on 16P, using the parameter optimized for 8P, is 81.5% higher than with the parameter optimized for 16P.

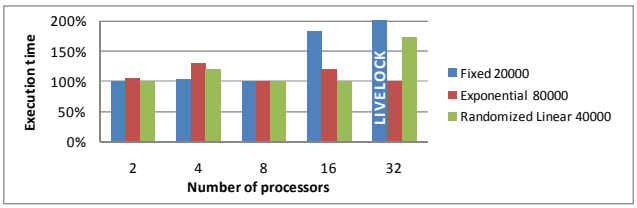

Figure 4. Execution time difference with parameters optimized for 8P

In order to compare the *abort handshake* scheme to the backoff policies, we use the optimal values obtained with our experiments for each policy and number of processors. No backoff policy can be pointed out as a winner, since all of them had close results and the best one depends on the particular case. The only policy that shows deviating results is Exponential, for LeeTM-IP on 16P and 32P.

Table 1 presents performance gains and energy savings of *abort handshake* against the three backoff policies. For 2P, our solution loses around 0.5% when compared to all backoff policies. It loses 4% when compared to the exponential backoff with 8P. In all other cases the *abort handshake* performs better than the other policies, reaching 20.25% of improvement over the best performing policy for 16P, the fixed backoff. The *abort handshake* for LeeTM–IP shows more modest speedups in general. The exception is the comparison with the exponential backoff with 16P and 32P, but the reason for these impressive results is the poor performance of exponential backoff for LeeTM-IP.

Except for the comparison with exponential backoff with 8P, where the *abort handshake* loses 33.7%, the energy results for LeeTM presented energy savings in almost all cases, except for a few cases with negligible losses. On the other side, even with modest performance improvements in most cases, LeeTM-IP has considerable energy savings, of up to 53.6% over the best backoff policy. Further work needs to be done to understand the sudden drop on the performance gains of the *abort handshake* for 32P.

VI. Conclusions and Future Work

This work presented an evaluation of performance and energy of a LogTM implementation in an NoC-based embedded MPSoC. The work also discussed problems with LogTM's backoff delay on abort, showing examples of how it can impact the overall system performance and how difficult it is to find a general, optimal solution for the delay. Different alternatives are analyzed, and a new solution is proposed based on a simple handshake between transactions. Results show that the proposed solution can improve performance up to 20% and provide energy savings up to 53% when compared to the best backoff policy among those evaluated in this work.

As future work, we plan to extend the evaluation of our solution with other benchmarks. In addition, we shall analyze the performance impact of different granularities of transactions and different cache block sizes.

References

[1] M.Herlihy and J.E.B.Moss. "Transactional Memory: Architectural Support for Lock-Free Data Structures". In International Symp. on Computer Architecture, May 1993.

[2] K.E.Moore, J.Bobba, M.J.Moravan, M.D.Hill, and D.A.Wood. "LogTM: Log-Based Transactional Memory". In Proc. of 12th International Symp. on High Performance Computer Architecture, February 2006.

[3] L.Kunz, G.Girão, and F.R.Wagner. "Evaluation of a Hardware Transactional Memory Model in an NoC-based Embedded MPSoC." In Proc. of the 23rd Symp. on Integrated Circuits and Systems Design (SBCCI '10). September 2010.

[4] W.N.Scherer III and M.L.Scott."Contention Management in Dynamic Software Transactional Memory". In Proc. of ACM PODC Workshop on Concurrency and Synchronization in Java Programs, July 2004.

[5] T.Moreshet, R.I.Bahar, and M.Herlihy. "Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks". In Workshop on Memory Performance Issues, February 2006.

[6] S.Sanyal, S.Roy, A.Cristal, O.S.Unsal, and M.Valero. "Clock Gate on Abort: Towards Energy-efficient Hardware Transactional Memory". In Proc. of the IEEE International Symposium on Parallel & Distributed Processing, May 2009.

[7] C.Ferri, A.Viescas, T.Moreshet, R.I.Bahar, and M.Herlihy. "Energy Implications of Transactional Memory for Embedded Architectures". In Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM'08), April 2008.

[8] C.Ferri, S.Wood, T.Moreshet, R.I.Bahar, and M.Herlihy. "Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems". In Proc. of International Conference on High-Performance Embedded Architectures and Compilers, January 2010.

[9] E.T.Silva Jr., D.Barcelos, F.R.Wagner, and C.E.Pereira. "An MPSoC Virtual Platform for Real-Time Embedded Systems". In JTRES'08 - 6th International Workshop on Java Technologies for Real-Time and Embedded Systems, September 2008.

[10] S.A.Ito, L.Carro, and R.P.Jacobi. "Making Java Work for Microcontroller Applications". Design & Test of Computers, IEEE, Sept. 2001.

[11] C.A.Zeferino, M.E.Kreutz, and A.A.Susin. "RASoC: a Router Soft-core for Networks-on-chip". In Proc. of Design, Automation and Test in Europe Conference and Exhibition, February 2004.

[12] A.C.S.Beck Filho, J.C.B.Mattos, F.R.Wagner, and L.Carro." CACO-PS: a General Purpose Cycle-accurate Configurable Power Simulator". In Proc. of 16th Symp. on Integrated Circuits and Systems Design, September 2003.

[13] H.-S.Wang, X.Zhu, L.-S.Peh, and S.Malik, "Orion: a Power-performance Simulator for Interconnection Networks". In Proc. of the 35th Annual ACM/IEEE International Symp. on Microarchitecture, November 2002.

[14] S.Wilton and N.Jouppi. "Cacti: An Enhanced Cache Access and Cycle Time Model". IEEE Journal of Solid State Circuits, May 1996.

[15] M.Ansari, C.Kotselidis, K.Jarvis, M.Luján, C.Kirkham, and I.Watson. "Lee-TM: A Non-trivial Benchmark for Transactional Memory". In ICA3PP, 2008.

TABLE I. Performance Gains and Energy Savings for the Abort Handshake Relative to the Backoff Policies

Performance Gains (LeeTM)	2P	4P	8P	16P	32P
Fixed	-0.51%	10.08%	6.50%	23.36%	0.03%
Exponential	-0.51%	7.07%	-4.00%	25.32%	6.96%
Randomized Linear	-0.48%	5.71%	5.31%	20.25%	12.25%

Energy Savings (LeeTM)	2P	4P	8P	16P	32P
Fixed	-0.04%	4.66%	9.86%	11.66%	2.07%
Exponential	-0.04%	-0.78%	-33.70%	25.42%	3.18%
Randomized Linear	-0.08%	3.54%	2.70%	27.38%	19.07%

Performance Gains (LeeTM_IP)	2P	4P	8P	16P	32P
Fixed	-1.40%	0.16%	2.17%	2.74%	-0.44%
Exponential	-1.40%	0.27%	1.66%	31.12%	55.27%
Randomized Linear	-1.40%	0.08%	1.56%	3.47%	0.82%

Energy Savings (LeeTM_IP)	2P	4P	8P	16P	32P
Fixed	-0.66%	6.72%	23.94%	53.60%	5.25%
Exponential	-0.66%	7.68%	20.04%	78.11%	70.06%
Randomized Linear	-0.05%	9.74%	24.67%	54.89%	8.41%