

A Hardware/Software Framework for supporting Transactional Memory in a MPSoC Environment

Cesare Ferri¹, Tali Moreshet², R.Iris Bahar¹, Luca Benini³, and Maurice Herlihy⁴

¹Brown University, Division of Engineering, Providence, RI 02912, USA

²Swarthmore College, Department of Engineering, Swarthmore, PA 19081, USA

³Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, 40136 Bologna, Italy

⁴Brown University, Department of Computer Science, Providence, RI 02912, USA

Abstract

Manufacturers are focusing on multiprocessor-system-on-a-chip (MPSoC) architectures in order to provide increased concurrency, rather than increased clock speed, for both large-scale as well as embedded systems. Traditionally lock-based synchronization is provided to support concurrency; however, managing locks can be very difficult and error prone. In addition, the performance and power cost of lock-based synchronization can be high. Transactional memories have been extensively investigated as an alternative to lock-based synchronization in general-purpose systems. It has been shown that transactional memory has advantages over locks in terms of ease of programming, performance and energy consumption. However, their applicability to embedded multi-core platforms has not been explored yet. In this paper, we demonstrate a complete hardware transactional memory solution for an embedded multi-core architecture, consisting of a cache-coherent ARM-based cluster, similar to ARM's MPCore. Using cycle accurate power and performance models for the transactional memory hardware, we evaluate our architectural framework over a set of different system and application settings, and show that transactional memory is a promising solution, even for resource-constrained embedded multiprocessors.

1 Introduction

Multi-core architectures have become pervasive in large-scale digital integrated systems. Manufacturers of general-purpose processors have essentially given up trying to increase clock speed, and instead are now focusing on multiprocessor-system-on-a-chip (MP-

SoC) architectures, in which multiple processors residing on a single chip communicate directly through shared hardware caches, providing increased concurrency instead of increased clock speed [8]. Integrated platforms for embedded applications are even more aggressively pushing core-level parallelism. SoCs with tens of cores are commonplace [23, 30, 32], and platforms with hundreds of cores have been announced [24].

In principle, multi-core architectures have the advantages of increased power-performance scalability (assuming on-chip interconnect scales well [7]) and faster design cycle time (by exploiting replication of pre-designed components). However, performance and power benefits can be obtained only if applications exploit a high level of concurrency. Indeed, one of the toughest challenges to be addressed by multi-core architects is how to help programmers expose application parallelism.

Embedded applications in multimedia, imaging, and communication, have a high degree of exposed thread-level parallelism, and this is one of the main reasons for the rapid diffusion of multi-core architectures in embedded systems [9]. However, managing concurrency is not easy. One major issue is how to synchronize concurrent accesses to memory. When multiple threads access a shared data structure, care must be taken to ensure that concurrent accesses do not interfere. Embedded multi-core architectures usually provide lock-based synchronization support for this purpose.

Lock-based synchronization relies on dedicated hardware. Atomic read-modify-write memory access is probably the most commonly deployed synchronization mechanism (e.g., it is used in the ARM MPCore architecture [5]). It requires support in the processor tile, in the communication fabric, and/or in the memory targets. For instance, MPCore relies on hard-

ware support for locked transactions in the system interconnect. Other architectures use semaphore target devices [22] that provide direct atomic read-modify-write support.

Embedded platforms are almost invariably deployed in tightly resource constrained contexts. Hence, the performance and power cost of synchronization is a serious concern that is further compounded by the issue of providing effective programming abstractions. In fact, managing locks is very difficult and error prone. Deadlocks are difficult to avoid, especially when systems have multiple resources. Moreover, even when the system is operating correctly, conditions like lock spinning may impose a significant overhead in power and performance, since they tend to flood the interconnect with useless transactions.

Transactional memories have been extensively investigated in general-purpose systems as an alternative to lock-based synchronization (e.g., [13, 16, 18, 10]). Transactional memory enables speculative execution of threads without acquiring locks, guaranteeing that transactions appear to execute atomically. Transactional memory has advantages over locks in terms of ease of programming, performance, and energy consumption [19, 25]. However, their applicability to embedded multi-core platforms, based on simple cores and memory system interfaces, has not been explored yet. The main objective of this paper is to demonstrate, for the first time, a complete hardware transactional memory solution for an embedded multi-core architecture, consisting of a cache-coherent ARM-based cluster, similar to ARM's MPCore. We have developed cycle accurate power and performance models for the transactional memory hardware, and we developed a lean software library that supports critical sections based on transactional memory. This hardware-software framework has been evaluated over a set of different system and application settings. Our analysis shows that, while transactional memory can provide clear performance advantages, careful consideration to hardware design is essential in order to meet the tight energy constraints of an embedded multiprocessor platform. Still, even with these tight energy constraints, our results show up to a 23% reduction in energy consumption, a 62% reduction in execution time, and a 75% improvement in energy delay product for an embedded application using transactional synchronization compared to the application using locks.

2 Background and Previous Work

Locks are the most common approach to synchronization. A lock indicates whether a shared data object is in use. A thread must acquire a lock before accessing the shared object, and release the lock after it is done. Locks may be implemented in many different ways, including semaphores or interrupts.

Despite their widespread use, locks have serious limitations. Coarse-grained locks, which protect relatively large amounts of data, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself becomes a source of

contention. Fine-grained locks, which protect smaller regions of memory, may appear more scalable, but they are difficult to use effectively and correctly. In particular, they introduce substantial software engineering problems, since the conventions associating locks with objects become more complex and error-prone. Locks also cause vulnerability to thread failures and delays: if a thread holding a lock is delayed by a page fault, or context switch, other running threads may be blocked. Locks also inhibit concurrency because they must be used conservatively: a thread must acquire a lock whenever there is a possibility of a synchronization conflict, even if such a conflict is actually rare. Finally, locks have a disadvantage in terms of energy consumption [19].

Transactional memory [13] is a speculative alternative to locks. Transactional memory can be implemented in hardware [16, 21, 25, 27] or in software [10, 11, 12, 15, 17, 28], or as a hybrid hardware-software combination [3, 18, 26]. In this paper, we investigate transactional synchronization specifically for embedded architectures. We focus on hardware transactional memory, since we feel this implementation is more appropriate for the needs of embedded systems. A more detailed overview of hardware transactional memory follows.

In transactional memory, each transaction is executed speculatively by a single thread without acquiring a lock. If the transaction completes without conflicting with another transaction, it commits, and its effects become permanent. Otherwise, if conflicts were detected during execution by the native hardware cache coherence protocol, the transaction aborts, its effects are discarded, and the transaction is restarted at a later time.

Until a transaction commits, its effects are not visible outside the transaction itself. To ensure such isolation, hardware transactional memory keeps tentative updates in a thread-local cache. If the transaction commits, these modified entries may be written back to memory. Data conflicts with other transactions will be detected by the native cache coherence mechanism. If a conflict is detected, the transaction is aborted, and its effects are discarded.

Transactional memory requires a checkpointing mechanism to support roll-back and re-issue of transactions in case of a conflict. Various checkpointing mechanisms have been proposed for recovering processor state. Many of these techniques periodically create a system-wide logical checkpoint of the system, which includes the state of the processor registers, memory values, and coherence permissions [29, 20]. For our purposes, it is sufficient to checkpoint the local registers of the processor that started the transaction.

Transactional memory improves ease of programming and performance of shared memory multiprocessors [13]. Hardware transactional memory provides a level of concurrency at least equivalent to the finest granularity locking, at a modest hardware cost [25]. Transactional memory also has an advantage over locks in terms of energy consumption [19], due to reduced contention and the absence of lock acquisition overhead.

In this paper, we take a first step to investigate the

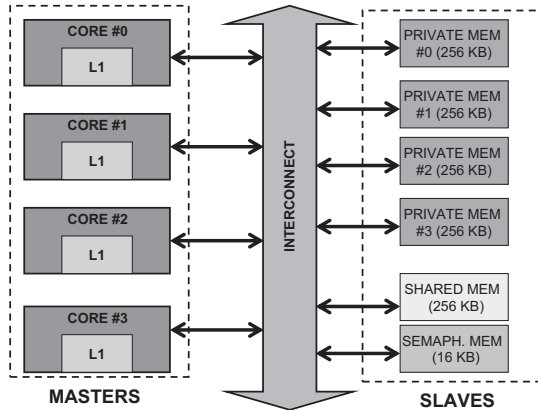


Figure 1. Example of the system configuration with 4 CPUs.

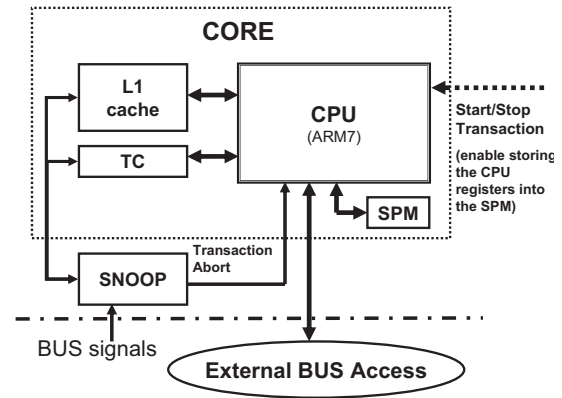


Figure 2. Architecture overview.

extent to which these advantages of transactional synchronization for general-purpose processors carry over to embedded MPSoC architectures.

3. Transactional Memory for Embedded Systems

3.1 Simulation Platform

We developed our architecture on top of the MPARM simulation framework [4, 14]. MPARM is a cycle-accurate, multi-processor simulator written in SystemC that provides high flexibility in terms of design space exploration. The designer can use the MPARM facilities to model any simple instruction set simulator with a complex memory hierarchy (supporting, for example, caches, scratchpad memories, and several types of interconnects). Finally, MPARM includes cycle-accurate power models for many of the simulated devices. The power models have been characterized by a $0.13\mu\text{m}$ technology provided by STMicroelectronics [31].

As shown in Figure 1, the adopted basic system configuration consists of a variable number of ARM7 cores (each having an 8KB direct-mapped L1 cache), a set of private memories (256KB each), one shared memory (256KB) and one bank (16KB) of memory-mapped registers which work as hardware semaphores. The interconnect is an AMBA-compliant communication architecture [1]. A Cache-coherence protocol (MESI) is also provided by snoop devices connected to the master ports. Note that while the private and shared memories are arbitrarily sized rather large (256KB) they do not significantly impact the performance or power of our system (as will be shown in Section 4).

3.2 Implementation

The basic idea behind the Transactional Memory working model is simple: each transaction is speculatively executed by the CPU and, if no conflicts with another transaction are detected, its effects become permanent (that is, the transaction *commits*). Otherwise, if conflicts are detected, its effects are discarded (that is, the transaction *aborts*), and the transaction is restarted. Hence, the support of the hardware that a Transactional Memory generally requires is 1) a safe location for storing/modifying transactional data, and 2) a rollback mechanism for re-executing the transaction.

We modeled our Transactional Memory after [13], and implemented a small (512B) fully-associative *Transactional Cache* (TC). The TC is accessed in parallel with the L1 cache; its main task is to manage all the read/write operations during a transaction. Preliminary experimental results suggest that most transactions are quite small, both in common benchmarks and even in the Linux kernel [2]. Therefore, in this paper, we do not consider transactions that would exceed the capacity of our TC.

Figure 2 provides an overview of the implemented architecture. To start a transaction, the CPU creates a local checkpoint by saving the contents of its registers into a small (128B) Scratchpad Memory (SPM) [6]. Note that during the time the CPU is writing into the SPM, the pipeline is stalled. In our case, moreover, the SPM must be carefully resized in order to contain the entire set of CPU registers.

During the execution of the transaction, two copies of accessed data will be stored in the TC, as shown in Figure 3. That is, two physical lines are maintained for each address: one line stores the backup copy of the data and the other contains the data modified during the transaction. This scheme requires adding 2 extra bits to the cache coherence tag vector. These bits will allow us to distinguish between the backup copy (marked with the T_COMMIT bit) and the modified data (marked with the T_ABORT bit). If neither bit is set, this indicates that the data in the line is not con-

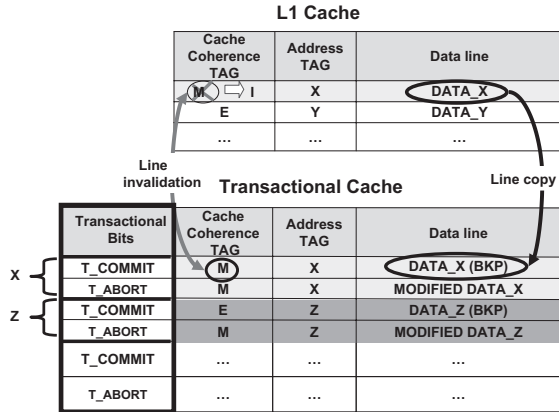


Figure 3. The TC maintains two copies for each line and is exclusive to the L1 cache.

tained within a transaction and therefore can be managed in the usual manner by the snoop device.

In case of a data conflict, the snoop device notifies the CPU with the *Abort.Transaction* signal which causes all the *T_ABORT* lines to be invalidated. Once the CPU receives the *Abort.Transaction* signal, it 1) stops the execution of the transaction, 2) restores the registers values by reading from the SPM, and finally 3) changes its status to “low power” mode. The CPU remains in this low power mode for some random backoff period after which it can begin re-executing the transaction. The range of the backoff period is tuned according to the conflict rate. That is, the first time a transaction conflicts it waits an initial random time period (< 100 cycles) before restarting. If the transaction conflicts again, the backoff period is doubled each time until the transaction completes successfully.

If there is no conflict, and the transaction commits, the TC invalidates the *T_COMMIT* lines and resets the *T_ABORT* bits of the tag vector. This results in the lines that were marked before as *T_ABORT*, now containing useful data for the rest of the system. As a consequence, the snoop device will manage these lines according to the native cache coherence protocol.

For correct implementation, the system needs a write-back (with write-allocate) cache policy, (e.g., all the writes have to be done within the TC). In addition, if the TC requires data contained in the L1 cache, the corresponding L1 lines will be copied to the TC and then invalidated in the L1 (guaranteeing non-overlapping data-sets). It is important to emphasize that the snoop device will continue to manage all the non-transactional valid lines (i.e., the lines which do not have the *T_COMMIT*, *T_ABORT*, or *INVALID* bits set) according to the standard cache-coherence protocol (the MESI scheme, in our case).

The software support for the transactions is provided by a library of special instructions that are invoked using macros. The macros produce a write

into a special memory-mapped location that controls the signals *Start.Transaction* and *Stop.Transaction*, as shown in Figure 2.

4 Experimental Results

In this section, we evaluate the power and performance benefits of using hardware transactional memory with an embedded multi-core system. For this purpose, we developed a parameterizable micro-benchmark that can be used to represent a number of common shared memory access patterns in embedded applications. The micro-benchmark consists of a sequence of atomic operations executed on a shared matrix, which is logically subdivided into overlapping regions. To ensure that concurrent accesses to shared data do not produce incorrect results, processors must obtain exclusive access to each region. Using a conventional scheme, we would associate a lock with each region to ensure that processors do not see inconsistent data at the boundaries of the matrix. Such a scheme is typically used in image-processing applications for embedded systems (such as plotters, printers, digital cameras). To run the micro-benchmark with transactional memory, we replaced the locks and critical sections defined above with transactions.

Embedded system applications may include varying computational workloads within critical sections, as well as outside of critical sections. To study the effect of such variations, we include four different configurations of our micro-benchmark, marked as *C1*, *C2*, *C3*, and *C4*, in Table 1. For example, *C2* denotes a micro-benchmark configuration with a medium computational workload inside critical sections and a light computational workload outside of them, meaning that transactions compose a large portion of the micro-benchmark.

Benchmark Configuration	Inside Crit. Section	Outside Crit. Section
<i>C1</i>	~60%	~40%
<i>C2</i>	~85%	~15%
<i>C3</i>	~20%	~80%
<i>C4</i>	~5%	~95%

Table 1. Micro-benchmark workload characterization.

Figure 4 shows the system energy of the different micro-benchmark configurations running on 4, 8, and 10 CPUs. Varying the number of CPUs allows us to explore the scalability of using transactions with increasing amounts of interconnect congestion. The system energy represents the sum of the energy of the cores, transactional caches (TC), L1 caches, scratchpad memories, RAMs and buses. Each group of bars represents a micro-benchmark configuration from Table 1 (*C1–C4*). The leftmost pair of bars in each group

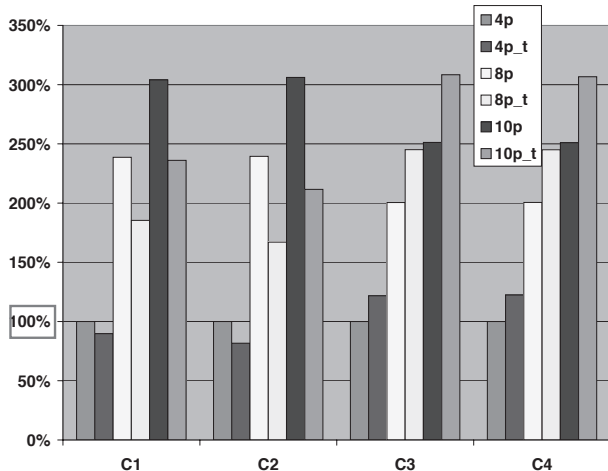


Figure 4. System energy of different micro-benchmark configurations with 4, 8, and 10 CPUs using locks (4p, 8p, 10p) and transactions (4p_t, 8p_t, 10p_t). The results within each group are normalized with respect to 4p.

represents the system energy when running on 4 CPUs and using locks (4p), or transactions (4p_t). The second and third pairs of bars in each group have a similar representation for 8 (8p, 8p_t) and 10 (10p, 10p_t) CPUs, respectively. Results within each group are normalized with respect to the first bar in the group (4p). Figure 5 uses similar notation to show the overall execution time for the different micro-benchmark configurations relative to the 4p configuration.

From looking at the first two groups of bars of Figures 4 and 5 (C1 and C2), we see that replacing locks with transactions reduces the system energy by 10%–31% and execution time by 42%–64%, relative to a configuration with the same number of processors, but using locks instead of transactions for synchronization. By contrast, the last two groups of bars (C3 and C4) show that although replacing locks with transactions has a negligible effect on the execution time, it has a noticeably negative effect on system energy. In these micro-benchmark configurations, most of the computation occurs outside the synchronized code, so transactions do not play a major role. The additional energy is consumed by the transactional cache described in Section 3. In the simple design considered here, the transactional cache is active for the duration of micro-benchmark execution since, once a transaction commits, it is possible that the only valid copy of certain data resides in this cache. It follows that every access to the L1 cache requires a parallel access to the transactional cache, even if no transaction is in progress. For micro-benchmark configurations C1 and C2, this additional energy is more than balanced by eliminating the extra energy consumed by locks, but not for configurations C3 and C4, which do much less

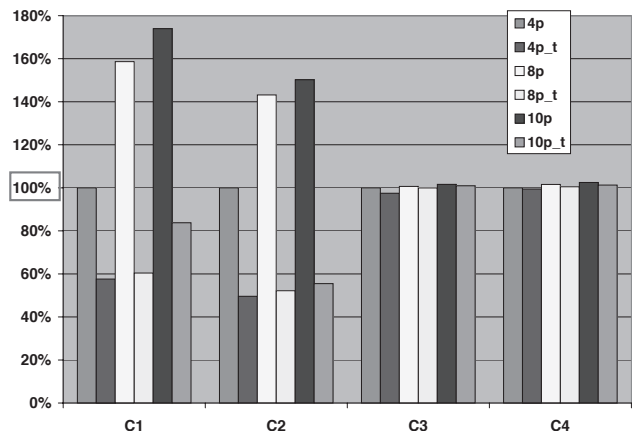


Figure 5. Execution time of different micro-benchmark configurations with 4, 8, and 10 CPUs using locks (4p, 8p, 10p) and transactions (4p_t, 8p_t, 10p_t). The results within each group are normalized with respect to 4p.

synchronization. These results suggest that it would be worth investigating more complex implementations that, for example, empty and power down the transactional cache when no transaction is in progress. Our preliminary experiments using such an implementation suggest that energy consumption could be improved substantially. For example, on a system with 8 CPUs, the energy consumption for benchmark configurations C3 and C4 would be reduced by 16% and 18% respectively, compared to results shown in Figure 4 for the 8p_t case. In other words, the overall energy consumption of the system would remain about the same whether a lock-based or transaction-based synchronization scheme were used for the C3 or C4 configurations. This makes sense since the fraction of time spent executing critical sections of the code is quite small.

To get a better understanding of the overall energy consumption, Figure 6 shows the energy distribution for the C1 micro-benchmark configuration with transactions and four CPUs.¹ As may be expected, the caches and CPUs are the most significant contributors to the system energy (the RAMs are on-chip, and consume a very small fraction of the energy). It is notable that the transactional caches (TC) and the CPUs consume comparable amounts of energy (20.65% and 19.47% respectively). The transactional cache is a fully-associative cache, which explains its high energy consumption. These results further emphasize the need to consider alternative schemes for operating the transactional cache, especially when running applications where transactions make up a relatively

¹Energy distribution for other configurations using transactions showed similar results.

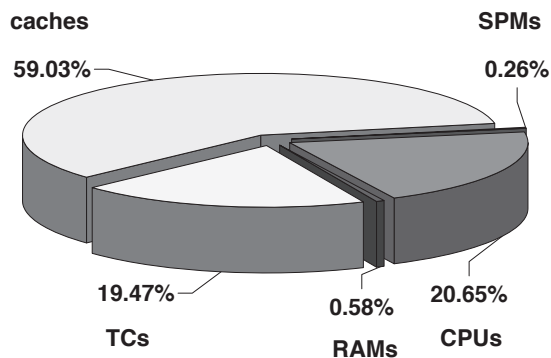


Figure 6. Energy Distribution for benchmark C1 with transactions and 4 CPUs (4p.t).

small portion of the total execution time. While our initial experiments suggest substantial energy benefits from being able to turn off the TC when the CPU is not executing transactional code, future work will investigate more precise hardware and software solutions for dealing with this issue.

To summarize our results, Figure 7 shows the energy delay product (EDP) of the different micro-benchmark configurations. Note from the figure that transactions introduce substantial improvements over locks for the C1 and C2 micro-benchmark configurations, mainly through reduction of overall execution time. For example, the C1 configuration with transactions and 8 CPUs (8p.t) obtains a system energy reduction of 23%, a 62% shorter execution time, and a 71% better EDP over the same configuration with locks (8p). Table 2 summarizes the experimental results for C1 and C2. Although increasing the time spent inside a transaction increases the probability for conflicts, transactional memory still manages to obtain higher throughput and lower energy.

Configuration	System Energy%	Exec. Time%	EDP%
C1			
4p.t vs. 4p	89.8%	57.6%	51.7%
8p.t vs. 8p	77.7%	38.1 %	29.6%
10p.t vs. 10p	77.7%	48.2%	37.4%
C2			
4p.t vs. 4p	81.7%	49.6%	40.5%
8p.t vs. 8p	69.7%	36.5%	25.5%
10p.t vs. 10p	69.1%	36.9%	25.5%

Table 2. Summary of C1, C2 results (lower values are better).

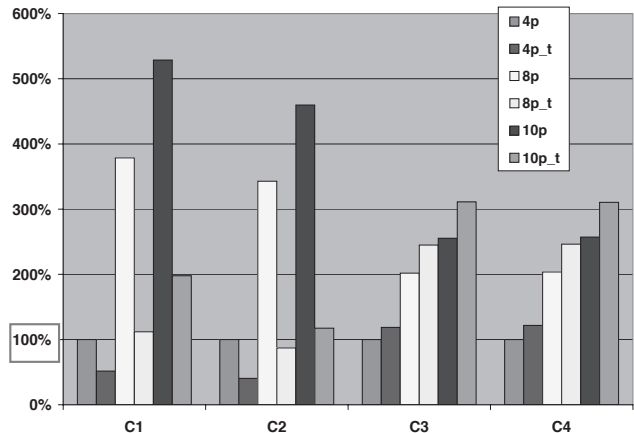


Figure 7. Energy*Delay Product (EPD), normalized with respect to 4p.

5. Conclusions

We view this work as a first step toward understanding the extent to which applications on embedded MP-SoC architectures can benefit from transactional synchronization. In particular, we are the first to present a transactional memory solution for a SoC platform technology that is modeled with cycle-accurate precision, and with accurate power models. We use frequency and power numbers and architectural assumptions that are appropriate for an embedded multiprocessor based on simple cores. Our results show that, while transactional memory can provide clear performance advantages, careful consideration to hardware design is essential in order to meet the tight energy constraints of an embedded system. This finding is substantially different from one obtained when analyzing transactional memory on a general purpose platform, since energy consumption is not so tightly constrained for these systems and the hardware to support transactional memory contributes a much smaller fraction to overall energy, as reported in [19]. Still, even with these tight energy constraints, our results show substantial improvement in terms of both energy and performance are possible using transaction-based synchronization on an embedded platform.

Significant work remains to be done to investigate other benchmarks and a wider range of architectural choices and hardware implementations. One interesting question is whether embedded applications will require new hardware mechanisms to support embedded applications. For example, little is known about how transactional synchronization interacts with patterns such as software pipelining, or with soft real-time constraints.

References

- [1] ARM Ltd. The advanced microcontroller bus architecture (AMBA) homepage. www.arm.com/products/solutions/AMBAHomePage.html.
- [2] C. S. Ananian, K. Asanovic, B. C. Duszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, pages 316–327, 2005.
- [3] C. S. Ananian, K. Asanovic, B. C. Duszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *International Symposium on High-Performance Computer Architecture*, February 2005.
- [4] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous MPSoC design space exploration. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1145–1150, 2006.
- [5] MPCore multiprocessor family. www.arm.com.
- [6] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.
- [7] L. Benini and G. D. Micheli. Networks on chip: a new SoC paradigm. *IEEE Computer*, 35(1), January 2002.
- [8] S. Borkar and et al. Platform 2015: Intel processor and platform evolution for the next decade. Technical report, Intel, 2005. White Paper.
- [9] G. Declerck. A look into the future of nanoelectronics. In *IEEE Symposium on VLSI Technology*, pages 6–10, 2005.
- [10] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2003.
- [11] T. Harris, S. Marlowe, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming (PPOP)*, 2005.
- [12] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Symposium on Principles of Distributed Computing*, July 2003.
- [13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, May 1993.
- [14] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *Design and Test in Europe Conference (DATE)*, pages 752–757, February 2004.
- [15] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. Technical Report TR 868, Computer Science Department, University of Rochester, May 2005.
- [16] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [17] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, New York, NY, USA, 1997.
- [18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *International Symposium on High-Performance Computer Architecture*, February 2006.
- [19] T. Moreschet, R. I. Bahar, and M. Herlihy. Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks. In *Workshop on Memory Performance Issues*, February 2006. in conjunction with HPCA.
- [20] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReVive/IO: Efficient handling of i/o in highly-available rollback-recovery servers. In *International Symposium on High-Performance Computer Architecture*, February 2006.
- [21] J. Oplinger and M. S. Lam. Enhancing Software Reliability with Speculative Threads. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [22] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonard, O. Benny, B. Lavigneur, D. Lo, G. Beltrame, V. Gagné, and G. Nicolescu. Parallel programming models for a multiprocessor platform applied to networking and multimedia. *IEEE Transactions on VLSI Systems*, 14(7):667–680, July 2006.
- [23] Philips nexperia platform. www.semiconductors.philips.com.
- [24] PC205 platform. www.picochip.com.
- [25] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *International Symposium on Computer Architecture*, June 2005.
- [27] P. Rundberg and P. Stenström. Speculative Lock Reordering: Optimistic Out-of-Order Execution of Critical Sections. In *International Parallel and Distributed Processing Symposium*, April 2003.

- [28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, Special Issue(10):99–116, 1997.
- [29] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, May 2002.
- [30] Nomadik platform. www.st.com.
- [31] STMicroelectronics. www.stm.com.
- [32] OMAP5910 platform. www.ti.com.