

Energy Implications of Transactional Memory for Embedded Architectures

Cesare Ferri

Division of Engineering, Brown
University
cesare@lems.brown.edu

Amber Viescas

Computer Science Department,
Swarthmore College
aviesca1@swarthmore.edu

Tali Moreshet

Engineering Department,
Swarthmore College
tali@swarthmore.edu

Iris Bahar

Division of Engineering, Brown University
iris@lems.brown.edu

Maurice Herlihy

Computer Science Department, Brown University
mph@cs.brown.edu

Abstract

Roughly ninety percent of all microprocessors manufactured in any one year are intended for embedded devices such as cameras, cell-phones, or machine controllers. We evaluate the energy-efficiency and performance of spin-locks and simple hardware transactional memory on embedded devices. In most cases, transactional memory provides both significantly reduced energy consumption and increased throughput (up to 30% and 66% improvement, respectively, compared to locks). Nevertheless, for benchmarks where synchronization is sufficiently rare, the energy costs of the underutilized transactional caching hardware can outweigh its benefits. In these cases, it is often effective to power down the transactional cache between transactions.

1. Introduction

Roughly ninety percent of all microprocessors manufactured in any one year are intended for electronic products such as cameras, cell-phones, or machine controllers. Such embedded devices may be inconspicuous, but they affect every aspect of modern life. Like their general-purpose counterparts, embedded devices are increasingly exploiting core-level parallelism. Multi-processor systems on-chip (MPSoCs) with tens of cores are commonplace [18, 21, 23], and platforms with hundreds of cores have been announced [19]. These trends mean that, in the medium term, system designers and software engineers must learn to make effective use of increasing parallelism.

Because many embedded devices run on batteries, *energy efficiency* is perhaps the single most important criterion for evaluating hardware and software effectiveness in embedded devices. We evaluate the energy-efficiency (and performance) of a number of synchronization mechanisms adapted for embedded devices. In addition to power limitations, embedded systems are also more resource-limited than general

purpose systems, especially with regard to smaller caches and memories. Here, we focus on providing simple hardware accelerators for common software synchronization patterns found in embedded platforms. Concurrency in embedded devices takes several forms. Here, we consider applications in which threads communicate through shared data structures. Traditionally, such data structures have been protected by mechanisms such as locks and monitors. More recently, approaches for general purpose systems have focused on transactional memory [10]. However, current research on transactional memory has focused exclusively on general-purpose processors. Its applicability to embedded multi-core platforms, based on simple cores and memory system interfaces, has yet to be explored. Natural questions are then, “Is transactional memory an attractive alternative to locks for embedded systems? Is it energy-efficient?”

General-purpose processors must run a wide variety of applications, so they tend to have fast processors and plenty of memory. In contrast, embedded processors run specialized applications and tend to provide substantially fewer resources. As a result, there are very different trade-offs to be made between hardware and software. For example, in a general-purpose processor, it makes sense to do as much as possible in software, while in a specialized, resource-constrained embedded processor, it is more reasonable to move functionality into special-purpose hardware. This constraint can have far-reaching effects on how transactional memory may be implemented in an embedded system. In this paper, we investigate the energy implications of hardware transactional memory for MPSoC architectures. We show that transactional memory can provide clear performance and energy advantages for synchronization-intensive applications compared to locks (up to 66% and 30% improvement, respectively). However, for benchmarks where synchronization is sufficiently rare, the energy costs of the

underutilized transactional caching hardware can outweigh its benefits. In response, we develop an adaptive transactional memory scheme that avoids this high energy consumption scenario by powering down the transactional cache between transactions.

2. Background and Previous Work

Embedded architectures such as the ARM processor offer multiple processors on a chip [7]. Taking full advantage of the parallelism provided by a shared memory multiprocessor requires support for synchronization, ensuring that concurrent accesses to shared memory can proceed without interference, which is traditionally accomplished by locks.

Some researchers have explored the energy implications of locks for multi-processor systems-on-a-chip. Loghi *et al.* [12] evaluated power-performance tradeoffs in lock implementations. Monchiero *et al.* [14] proposed a *synchronization-operation buffer* as a high-performance yet energy-efficient means of handling spin locks by reducing memory references and network traffic.

Hardware transactional memory has been extensively investigated in general-purpose systems as an alternative to lock-based synchronization (for example, [10, 13, 20, 15]). To our knowledge, Moreschet *et al.* [16, 17] were the first to consider the energy implications of transactional memory on general-purpose multiprocessors. This paper differs from this earlier work by focusing on an embedded architecture [11] rather than a general-purpose architecture. Later work by Ferri *et al.* [6] has considered transactional memory within an embedded system framework; however, their study was limited to a single microbenchmark. In this paper, we extend the work of [6] by considering a broader set of benchmarks, and considering, for the first time, the possibility of powering down an underutilized transactional cache.

Recent work on transactional memory such as [15] focused on hybrid mechanisms that combine hardware and software, falling back on software for transactions that do not fit in the cache. Because we are targeting resource-poor embedded systems, we cannot afford such elaborate mechanisms. Instead, we focus on simple hardware synchronization mechanisms. For instance, on a transaction overflow, we stall all other transactions and allow the overflowed transaction to complete (TCC [9] uses a similar approach). Because transaction sizes for embedded applications are likely to be known in advance, we expect overflow to be extremely rare in practice. Blundell *et al.* [5] investigate a similar approach. They introduce a permissions-only cache that increases the size of transactions that fit in the cache and reduces the likelihood of an overflow. That technique could complement the approaches investigated here.

3. Synchronization Support for MPSoCs

We developed our architecture on top of the MPARM simulation framework [3, 11]. MPARM is a cycle-accurate,

Figure 1. Example Configuration with four CPUs.

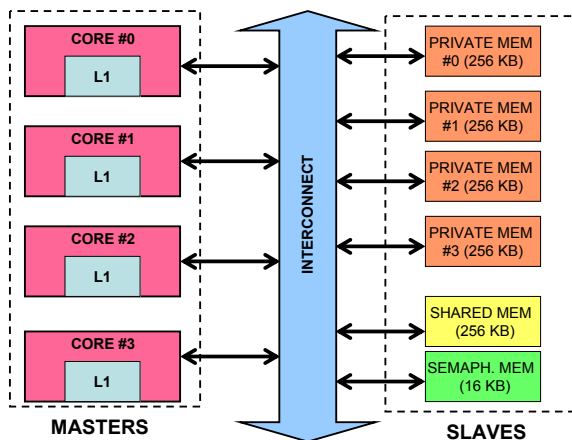
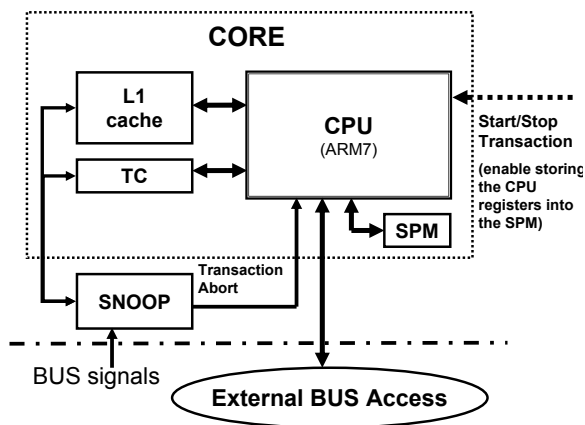


Figure 2. Architecture Overview.



multi-processor simulator written in SystemC [8] that provides 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 multiple types of interconnects). MPARM also includes cycle-accurate power models for many of its simulated devices. The power models reflect a $0.13\mu\text{m}$ technology provided by STMicroelectronics [22].

As shown in Figure 1, the basic system configuration consists of a variable number of ARM7 cores (each with an 8KB direct-mapped L1 cache), a set of private memories (256KB each), a single shared memory bank (256KB), and one bank (16KB) of memory-mapped registers serving 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 sized arbitrarily large (256KB each), they do not

significantly impact the performance or power of our system (as will be shown in Section 4).

We model our transactional memory after [10] and implement a small (512B) fully-associative *Transactional Cache* (TC). A fully-associative configuration allows for maximum utilization of the small cache, but, as we show in Section 4, comes with a significant cost in energy consumption. Note that if the TC were not fully associative, a miss due to a line conflict may need to be handled similarly to a data conflict with another transaction; i.e., the transaction will have to abort. The TC, accessed in parallel with the L1 cache, manages all read/write operations during a transaction. If a transaction exceeds the capacity of our TC, the system will stop all other processors that are currently executing transactions, and allow the transaction under question to complete using the rest of the memory hierarchy (e.g., L1 and L2). In particular, a special device (i.e. the TC-overflow manager) is in charge of synchronizing all the operations required to handle such a scenario. Whenever an overflowing transaction occurs, the TC-overflow manager enables the *Abort_and_Stop* signal that forces all the processors to halt. Only after all the processors have been blocked (and this can be easily verified by the TC-overflow manager, since the CPUs send back an ACK signal as soon as they stop their execution) the CPU with the overflowing transaction is allowed to proceed. While this is not the most efficient implementation to handle large transactions, we note that prior experiments have shown that large transactions are quite rare, both in common benchmarks and even in the Linux kernel [2]. Therefore such scenarios will be quite infrequent.

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) [4]. Note that while the CPU is writing into the SPM, the pipeline is stalled. In our implementation, the SPM must be carefully sized to provide storage for the entire set of CPU registers.

In case of a data conflict, the snoop device notifies the CPU with the *Abort_Transaction* signal, causing all speculatively-updated lines to be invalidated. When the CPU receives the *Abort_Transaction* signal it 1) halts execution of the transaction, 2) restores the original register values by reading from the SPM, and 3) changes its status to a “low power” mode. The CPU remains in this low power mode for a 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. The first time a transaction conflicts, it waits an initial random period (< 100 cycles) before restarting. If the transaction conflicts again, the backoff period is doubled, and will continue to double each time until the transaction completes successfully.

Software support for transactions is provided by a library of special instructions invoked using macros. These macros produce a write operation into a special memory-mapped

location that controls, for example, the *Start_Transaction* and *Stop_Transaction* signals shown in Figure 2.

As we will show in Section 4, experimental results suggest that transactional synchronization proposed here is effective on embedded platforms only for synchronization-heavy workloads, and that it may not be worthwhile to adapt the architecture to workloads that do not perform much synchronization. To address this challenge, we developed an enhancement to the transactional hardware that, when triggered, flushes the contents of the TC back into the traditional cache hierarchy and powers down the TC until the start of the next transaction implicitly re-enables its use. Doing so can save energy consumed in searching the TC in parallel with the L1 cache while outside of a transaction. Experimentation with multiple policies for triggering this TC shutdown revealed that, while static software hints may prove useful for certain transaction patterns, a policy of aggressively shutting down the TC at the end of each and every transaction generally conserves more energy. This “aggressive shutdown” policy also has the advantage of being a dynamic solution which can be applied to any program without programmer intervention or recompilation.

Recall that the TC was added to the cache hierarchy such that it is accessed in parallel with the L1 cache. When TC lines are written back, either during normal operation outside of a transaction, or triggered by a TC shutdown, the obvious approach would be to write back to external L2 memory. This is similar to an L1 writeback, and does not require additional architectural support. This approach, however, has a few drawbacks. These writebacks increase bus traffic, which leads to an increase in latency and energy consumption, and potentially delays other bus traffic. Moreover, if recently written back data is used by the processor outside the transaction, the L2 writeback will require fetching it back from L2, which will in turn double the amount of wasteful bus traffic. Due to these drawbacks, we also implemented an aggressive shutdown policy that writes data back to the local L1 cache instead of L2. Writebacks to L1 reduce bus traffic and enable re-use of data outside of a transaction, but also require additional architectural support for properly handling dirty lines encountered in the L1 during the flush process. In addition, the snoop device cannot use lines that are in the process of being replaced. To complicate matters further, an L1 writeback can potentially trigger an L2 writeback. However, as we show in Section 4, some of our proposed transactional memory configurations show a clear advantage of aggressive shutdown under certain workloads. The optimal configuration for each workload (including the writeback policy) could therefore be selected dynamically at run-time.

4. Experimental Results

This section presents the results of evaluating the energy and performance of different synchronization mechanisms in the

MPPARM simulation framework. We include a varied set of benchmarks that represent applications that require mutual exclusion shared-memory synchronization. The benchmarks are listed in Table 1.

Table 1. The evaluation benchmarks.

Benchmark	Applications
skip-list	memory management
red-black tree	memory management
matrix micro-benchmark	image processing

These are representative of embedded applications that perform data exchanges between data structures in shared memory. Linked lists and red-black trees are used in memory management. The matrix microbenchmark is representative of image-processing applications typically found in plotters, printers, and digital cameras, where synchronization is required to allow data to be exchanged within a matrix.

The simulator allows us to vary the number of cores as well as the percentage of shared data, providing two ways to gauge how benchmarks behave with different volumes of core interconnect congestion. To test transactional memory, we replace critical sections with transactions, replacing lock acquisition and release calls with *Start_Transaction* and *Stop_Transaction* macros, calling on the transactional hardware to ensure proper synchronization. We also use the simulator to test the effectiveness of various shutdown policies. For now, the default shutdown policy is never to shut down the transactional cache.

For testing matrix applications, we use a parameterizable micro-benchmark which can be tuned to four different scenarios: C5, C20, C60, and C85. They represent benchmarks respectively spending 5%, 20%, 60%, and 85% of their total execution cycles within critical sections. The micro-benchmark consists of a sequence of atomic operations executed on a shared matrix, logically subdivided into overlapping regions. To ensure that concurrent accesses to overlapping regions do not conflict, processors obtain exclusive access to each region, using either locks or transactions.

The difference between the micro-benchmark configurations with 5% and 20% of total execution spent in a critical section is minor, both in terms of contribution to delay and energy consumption. Both result in increased energy consumption in exchange for a small improvement in performance when using transactions.

Figure 3(a) shows the total system energy of the matrix micro benchmark. This energy includes the energy used by processor cores, transactional caches, L1 caches, scratchpad memories and RAM. The graph shows the effect of increasing the proportion of time spent inside a critical section when different number of cores are available.

As either the number of cores or the proportion of time spent in the critical section increases, transactional memory has an increasing advantage over locks. This advantage is

due to an increase in the amount of synchronization traffic required to maintain the consistency of the memory shared between the cores, which justifies the additional energy consumed by the transactional cache. When using locks, more cores and larger critical sections would likely result in an increase in the number of conflicts on the lock, resulting in an increase in overall execution time. Moreover, when an application spends only a small proportion of its run time in critical sections, aggressively shutting down the cache reduces the overall energy consumed. Conversely, if the TC is heavily used, the aggressive shutdown policy may be counterproductive.

Similarly, Figure 3(b) shows the overall execution time to complete the matrix micro-benchmark. As might be expected, the transactional memory system offers little to no improvement for benchmarks that spend the majority of their execution outside of critical sections (that is, only 5% or 20% of the time accessing shared data). With so little time spent performing shared memory operations, the choice of synchronization mechanism has little effect on throughput. In addition, with few cores in use, one cannot exploit transactional memory’s potential for increased parallelism. However, when the proportion of execution inside the critical section is increased to 60% and 85%, the benefit of transactional synchronization become evident. In fact, performance increases slightly as the critical section increases, in contrast to the decrease in performance seen in lock-based systems. Finally, the graph shows the scalability of transactional memory, as performance is improved with increasing number of cores. Workloads with many cores and large proportion of the work in critical sections are also likely to result in a high conflict rate. As an example, let’s look at the last set of bars on the right of Figures 3 (a) and (b), with 8 cores and 85% of the work inside critical sections. Indeed, in this configuration, 35% of transactions are aborted. However, this relatively high abort rate is more than compensated for by the reduced bus occupancy of transactions. Here, the bus is busy up to 90% of the time with locks, vs. less than 10% with transactional memory, resulting in the throughput and energy advantage discussed above.

Returning to the system energy graph, note that improvements to execution time can come at a price. Figure 3(a) shows that with a small proportion of critical sections or few cores, transactional memory can consume more energy than locking. However, closer inspection reveals that transactional memory harms energy efficiency only in those scenarios where transactions have little to no impact on cycle counts. In these cases, transactional memory cannot help reduce bus traffic significantly. Adding aggressive shutdown to a transactional memory system comes with its own trade-offs. Although energy consumption decreases relative to that of transactional memory with no shutdown, so does performance. One way to balance this trade-off is to consider the energy-delay product.

Figure 3. System energy and execution cycles of the matrix micro-benchmark using transactions, normalized to an equivalent system using locks. Note that negative percentages indicate energy/performance *advantage* relative to locking.

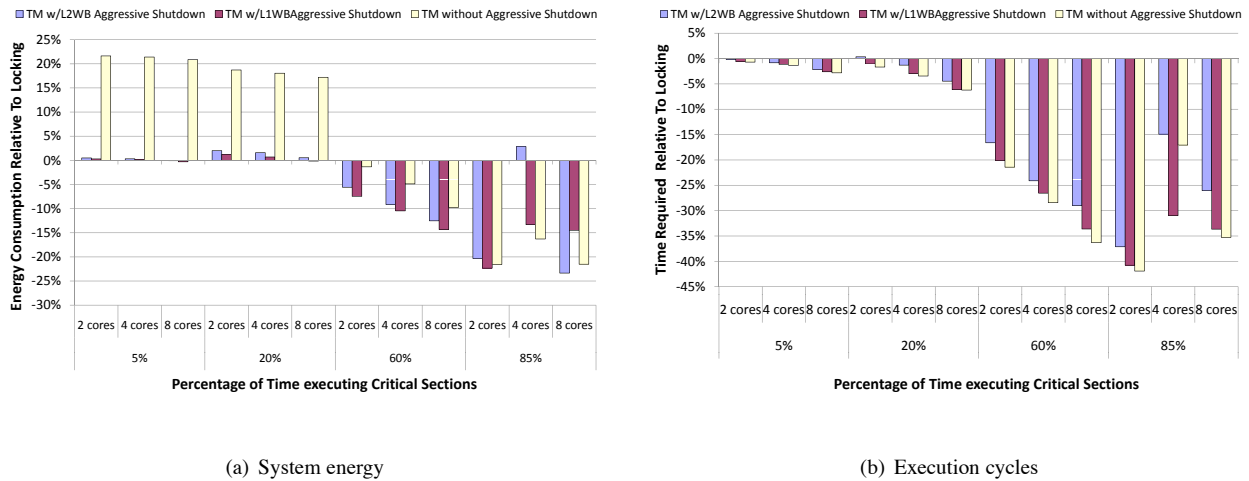
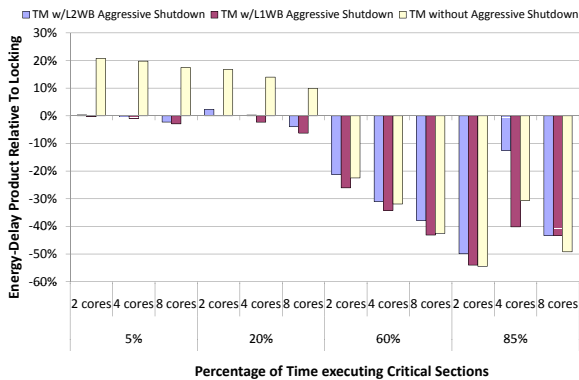


Figure 4. Energy-Delay product of the matrix micro-benchmark using transactions, normalized to an equivalent system using locks. Note that negative percentage indicate energy-delay *improvement* relative to locks.



The energy-delay results for these benchmarks are summarized in Figure 4. The graph shows that the L1 Write-Back Aggressive Shutdown policy almost always achieves better results than the other two configurations; only for the highest degree of synchronization is this not always the case. These results justify the extra architectural support needed to flush the contents of the TC to the L1 cache.

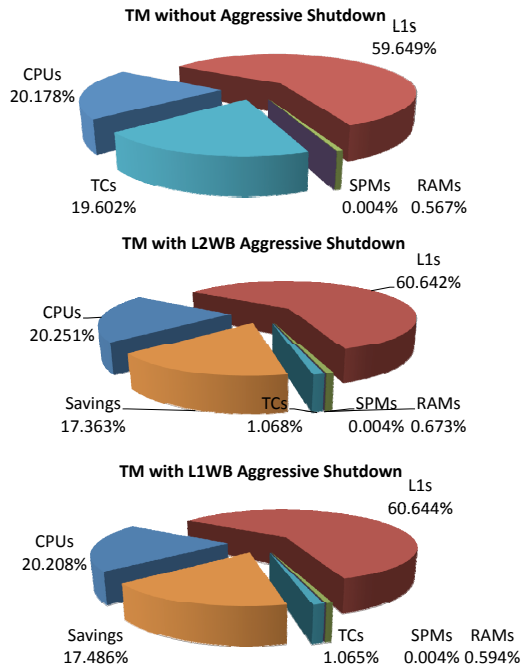
To get a better understanding of the energy overhead of the transactional implementation, we take a closer look at the energy distribution among the various components of the MPSoC system. Figure 5, showing the distribution of energy consumption among the various components of the MPSoC system, suggests that aggressive shutdown can be very effective for certain applications. The top chart shows the matrix

micro-benchmark on a system with transactions enabled, four processors, and running the least synchronization-intensive scenario (5%). While nearly 60% of system energy usage is that of the traditional L1 caches, almost 20% is consumed by the TC (which does nothing to reduce execution time under this workload). The remainder is consumed primarily by the processors themselves, while on-chip RAM and scratchpad memories contribute vanishingly small percentages to the overall system energy usage. When our aggressive policies for shutting down the TC are enabled under the same conditions (the two bottom charts of Figure 5), energy usage attributed to the TC drops to less than 2%, and the absolute energy consumed by the system is reduced by nearly 17%. This result approaches the energy (and execution time) statistics of the locking solution. Note also that for this particular workload, the energy saved is almost the same for both the aggressive shutdown mechanisms (L1WB vs. L2WB); however, this is not the case for all our tested configurations. Through targeted application of this aggressive shutdown policy to applications exhibiting low levels of synchronization traffic, the system’s energy consumption can be reduced to within 5% of locking while maintaining a negligible impact on execution cycles.

Continuing research in the area of TC shutdown seeks to identify effective methods by which the shutdown policy might be *dynamically* enabled and disabled in response to real-time analysis of the executing embedded application. In this manner, a MPSoC implementation could offer all the benefits of transactional memory in high traffic scenarios while still retaining the flexibility to revert to “lock-like” performance and energy behavior under less synchronization-intensive conditions.

The link-list applications are tested using standard skip-list and red-black tree benchmarks. The skip-list benchmark

Figure 5. Energy distribution for the matrix micro-benchmark with 5% of execution inside a critical section, with 4 processors, both with and without an aggressive transactional cache shutdown policy.



workload consists of 90% database lookups, 9% insertions, and 1% deletions, which is typical of most applications. For different experiments, the time spent inside critical sections can be 30%, 60%, or 90%, and the number of cores can be 2, 4, or 8.

The red-black tree benchmark and skip-list micro-benchmark show similar trends, so we will summarize their energy and performance results, showing only the energy-delay product for comparison. The EDP (Energy-Delay Product) results for the skip-list benchmark are reported in Figure 6. Figure 6 shows that, when only a small portion of the application has critical sections, aggressive shutdown is comparable to no shutdown. For larger sections, aggressive shutdown does not pay off. In the red black-tree benchmark, differences between transactional memory and locking are most noticeable with many cores and a large critical section. However, the red-black tree benchmark benefits more in terms of performance and energy when using transactional memory compared to the skip-list benchmark, particularly when aggressive shutdown is not used. This result is summarized in Figure 7 where we show the energy-delay product compared to locks for the red-black tree. In this scenario, transactional memory is more effective than locking, although using ag-

Figure 6. Energy-Delay product of the skip-list benchmark using transactions, normalized to an equivalent system using locks. Note that negative percentage indicate energy-delay *improvement* relative to locks.

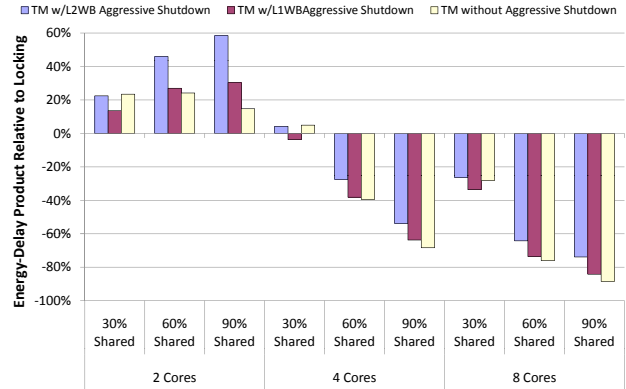
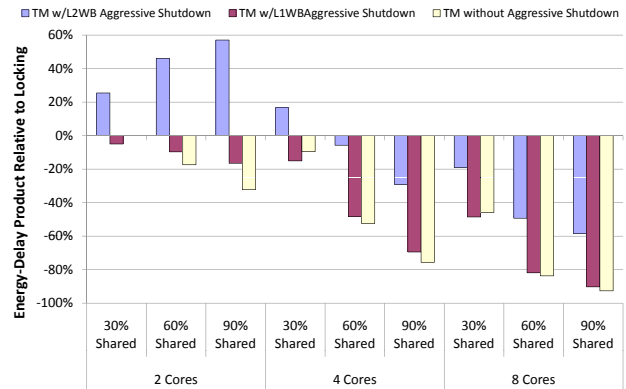
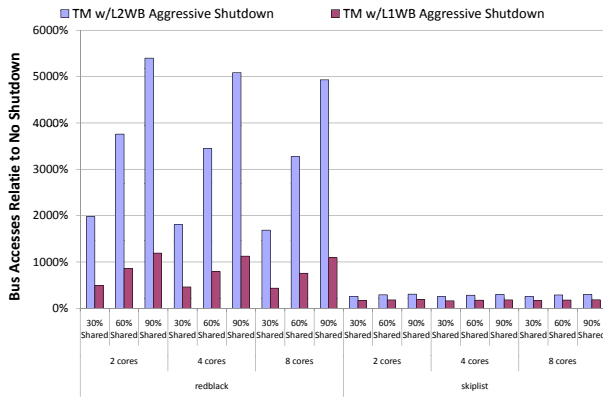


Figure 7. Energy-Delay product of the the red-black Tree benchmark using transactions, normalized to an equivalent system using locks. Note that negative percentage indicate energy-delay *improvement* relative to locks.



gressive shutdown is not beneficial, even for small critical sections, because the red-black tree benchmark produces little overall bus traffic (unlike the skip-list benchmark). Here, aggressive shutdown increases bus traffic dramatically, resulting in a large time delay and only a small increase in energy efficiency. Finally, Figure 8 shows the red-black and skip-list bus access rates for both the aggressive shutdown policies (L2WB vs L1WB) relative to no shut down. The graph confirms what we expected: writing back in L1 cache hides the inherent performance penalty due to the TC line charging/discharging operations.

Figure 8. Bus access rates for the red-black and skip-list benchmarks using L2WB aggressive shutdown and L1WB aggressive shutdown relative to no shutdown.



5. Conclusions

This paper provides initial insights into implementing an energy efficient transactional memory design targeting specifically for embedded systems. While transactional memory has been widely investigated for general purpose systems, we are the first to investigate the energy costs of transactional memory on an embedded platform using a cycle-accurate simulator. Since embedded applications have been shown to contain high degrees of concurrency, it seems natural to implement mechanisms such as transactional memory in order to better exploit this concurrency. Our results show that because embedded systems are constrained very differently compared to general purpose systems, implementing transactional memory using established mechanisms will not necessarily lead to an energy-efficient solution, and may even be counterproductive in terms of performance. To address this issue, we developed an enhancement to the transactional hardware, that, when triggered, flushes the contents of the TC back into the traditional cache hierarchy. Using such an approach can lead to a 17% savings in energy over a traditional transactional memory implementation. We also explored the trade offs of writing back the TC data to either the L1 or L2 cache. While a flush to the L2 cache is more straightforward to implement, depending on the application, this may cause increased bus traffic later on when the data may be needed again (either outside of or within another transaction). Finally, for future work, we plan to evaluate the tradeoffs of having a dedicated transactional cache versus storing transactions in the L1 cache. We would also like to conduct experiments on a wider range of embedded applications. As we have discovered with the current set of benchmarks, no single synchronization approach is likely to be energy-efficient for all applications. Testing our architecture

on a broader range of applications will allow us to develop more complete solutions.

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] 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.
- [4] 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.
- [5] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *International Symposium on Computer Architecture*, June 2007.
- [6] C. Ferri, T. Moreshet, R. I. Bahar, L. Benini, and M. Herlihy. A hardware/software framework for supporting transactional memory in a mpsoC environment. *SIGARCH Comput. Archit. News*, 35(1):47–54, 2007.
- [7] J. Goodacre and A. N. Sloss. Parallelism and the ARM instruction set architecture. *IEEE Computer*, 38(7), July 2005.
- [8] T. Grtker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002. ISBN 1402070721.
- [9] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). *SIGOPS Oper. Syst. Rev.*, 38(5):1–13, 2004.
- [10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, May 1993.
- [11] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MP-SoC environment. In *Design and Test in Europe Conference (DATE)*, pages 752–757, February 2004.
- [12] M. Loghi, M. Poncino, and L. Benini. Cache coherence tradeoffs in shared-memory MPSoCs. *ACM Transactions on Embedded Computing Systems*, 5(2):383–407, May 2006.
- [13] 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.
- [14] M. Monchiero, G. Palermo, C. Silvano, and O. Villa.

Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *IEEE/ACM Design Automation and Test in Europe*, Munich, Germany, April 2006.

- [15] 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.
- [16] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *International Symposium on Low Power Electronics and Design*, August 2005.
- [17] T. Moreshet, 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.
- [18] Philips nexperia platform. www.semiconductors.philips.com.
- [19] PC205 platform. www.picochip.com.
- [20] 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.
- [21] Nomadik platform. www.st.com.
- [22] STMicroelectronics. www.stm.com.
- [23] OMAP5910 platform. www.ti.com.