

Energy Efficient Synchronization Techniques for Embedded Architectures

Cesare Ferri, R. Iris Bahar
Division of Engineering
Brown University
Providence, RI 02901

Tali Moreshet, Amber Viescas
Engineering Department
Swarthmore College
Swarthmore, PA 19081

Maurice Herlihy
Computer Science Department
Brown University
Providence, RI 02901

ABSTRACT

We evaluate the energy-efficiency and performance of a number of synchronization mechanisms adapted for embedded devices. We focus on simple hardware accelerators for common software synchronization patterns. We compare the energy efficiency of a range of shared memory benchmarks using both spin-locks and a simple hardware transactional memory. In most cases, transactional memory provides both significantly reduced energy consumption and increased throughput. We also consider applications that employ concurrency patterns based on semaphores, such as pipelines and barriers. We propose and evaluate a novel energy-efficient hardware semaphore construction in which cores spin on local scratch-pad memory, reducing the load on the shared bus.

Categories and Subject Descriptors

C.1.4 [Computer Systems Organization]: Processor Architectures—*Parallel Architectures*

General Terms

Performance

Keywords

Embedded Systems, Transactional Memory

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 [16, 19, 21], and platforms with hundreds of cores have even been announced [17]. 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. We first 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* [9]. In this paper, we compare the energy efficiency of a range of shared memory benchmarks using both spin-locks and a simple hardware transactional memory. Not surprisingly, the results depend on the application. In most cases, however, transactional memory provides both significantly reduced energy consumption and increased throughput. 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.

We next consider applications that employ concurrency patterns based on *conditional synchronization*, where one or more threads wait for others to “catch up”. For example, in a *pipelined* application, one thread is in charge of each stage of the pipeline. Because different pipeline stages may need varying amounts of time, threads communicate by bounded FIFO buffers. If a thread that gets too far ahead of its predecessor tries to enqueue an item to a full buffer, it blocks, as does a thread that tries to dequeue from an empty buffer. Pipelining is often used by graphics and networking applications found in embedded systems. Similarly, in applications employing *barriers*, the computation is run in phases, and barriers ensure that no thread starts the next phase until all have finished the current phase. Barriers are common in numeric and scientific applications, such as fast Fourier Transforms (FFT), also found in embedded applications.

Pipelining and barrier software are easily implemented using *semaphores*. Naïve semaphore implementations are not energy-efficient, mostly because of energy-expensive bus traffic. We propose and evaluate an energy-efficient hardware semaphore construction in which cores spin on local scratch-pad memory, and communicate directly, bypassing the shared bus. Moreover, powering down waiting cores can substantially increase the energy efficiency of applications that employ pipelines or barriers, with a negligible (or very modest) effect on throughput.

2. BACKGROUND AND PREVIOUS WORK

Taking full advantage of the parallelism provided by embedded architectures such as the ARM processor [7] requires support for synchronization in order to ensure that concurrent accesses to shared memory can proceed without interference.

Researchers have explored the energy implications of locks for multi-processor systems-on-a-chip. For example, Loghi *et al.* [12] evaluated power-performance tradeoffs in lock implementations. Monchiero *et al.* [13] proposed a *synchronization-operation buffer*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GLSVLSI'08, May 4–6, 2008, Orlando, Florida, USA.
Copyright 2008 ACM 978-1-59593-999-9/08/05 ...\$5.00.

as a high-performance yet energy-efficient spin lock implementation that reduces memory references and network traffic.

Hardware transactional memory has been extensively investigated in general-purpose systems as an alternative to lock-based synchronization (e.g. [9, 14]). To our knowledge, Moreshet et al. [15] were the first to address the energy implications of transactional memory on general-purpose multiprocessors. Later work by Ferri et al. [6] considered transactional memory within an embedded system framework. We extend the above work with a broader set of benchmarks, and by considering, for the first time, the possibility of powering down an underutilized transactional cache.

Since we are targeting resource-poor embedded systems, 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 [8] 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; using a permissions-only cache, they increase the size of transactions that fit in the cache and reduce the likelihood of an overflow. This technique could complement the approaches investigated here.

Conditional synchronization is another common form of synchronization, used for pipelined parallelism and for applications using barriers. An energy-efficient barrier implementation was previously proposed for general purpose processors. The *thrifty barrier* proposed by Li et al. [10] reduces the energy of spinning on a barrier by using a software predictor to decide in which low-power mode to put early-arriving threads, and a modified cache controller wakes up the thread in time to resume execution.

Embedded systems often employ some type of semaphore for synchronization [7, 18]. In this paper, we describe a hardware semaphore that has the same design goals as the “distributed” hardware semaphore proposed by Poletti et al. [18]; however, ours is implemented somewhat differently. Their approach stores shared data in private scratchpad memories, following a message-passing synchronization protocol. In our approach, the inter-processor communication relies on writes and reads in the shared memory. Further, we also enhanced the architecture and implemented barriers through semaphores. Our semaphore implementation works in parallel with the transactional cache so applications can make use of both mechanisms.

3. MPSOC SYNCHRONIZATION SUPPORT

We developed our architecture on top of the MPARM simulation framework [3, 11]. MPARM is a cycle-accurate, multi-processor simulator written in SystemC 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 MPARM also includes cycle-accurate power models for many of its simulated devices. The power models reflect a 0.13 μ m technology provided by STMicroelectronics [20].

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. An AMBA-compliant communication architecture [1] is used for the interconnect. 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. Our hardware implementation of transactional memory and conditional synchronization are described next.

Figure 1: Example configuration with 4 CPUs.

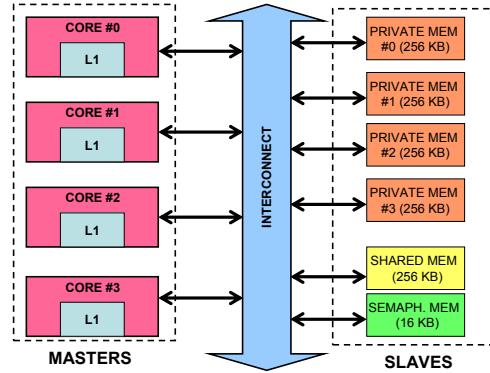
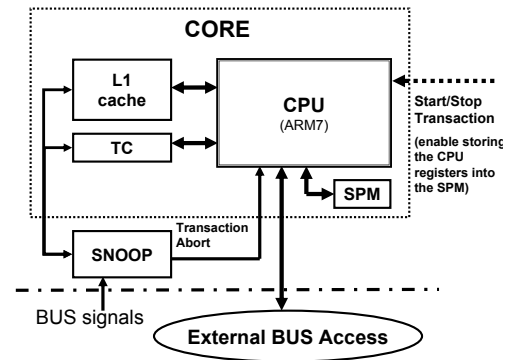


Figure 2: Architecture overview.



3.1 Transactional Memory

We model our transactional memory after [9] and implement a small (512B) fully-associative *Transactional Cache* (TC). 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 overflowing transaction to complete using the rest of the memory hierarchy. 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 may not be the most efficient way to handle transaction overflow, prior work suggests that large transactions are rare [2].

Figure 2 provides an architectural overview. 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.

If a data conflict occurs, the snoop device notifies the CPU by asserting the *Abort_Transaction* signal, causing all speculatively-updated lines to be invalidated. Once the CPU receives this 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 signals *Start_Transaction* and *Stop_Transaction* 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 outside of a transaction. We employ a policy of aggressively shutting down the TC at the end of each and every transaction, which we found to be optimal policy for triggering TC shutdown. 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.

3.2 Conditional Synchronization

To address pipelining, which is commonly used by embedded applications to achieve higher throughput, we extended our transactional memory architecture with a mechanism based on special distributed semaphores. Similar to [18], the semaphores are placed inside the cores, and are connected to an interrupt controller. Each internal semaphore is also attached to the bus through a slave port, and is mapped in a portion of memory visible to all the other processors. However, as opposed to the message-passing scheme described in [18], in our design the shared memory is the only inter-processor communication channel. The proposed method relies on ad-hoc atomic instructions, which are invoked from inside the transaction’s code. In particular, one instruction reads and decrements the value of the consumer semaphore, and forces the transaction to abort if the read value is not strictly positive. The rollback and backoff schemes are handled exactly as we described for transactions. At the re-execution of an aborted transaction, the CPU will restore the semaphore’s state by incrementing its value.

We enhanced the polling strategy used by semaphores with a more efficient method based on interrupts. We call this method the *irq policy*. When using the *irq policy*, the semaphore sends an interrupt request to the core as soon as its value changes from zero to one. In a producer-consumer pattern, for example, the consumer processor will be forced to wait in a low power mode until the producer processor increments the consumer’s semaphore. This differs from the conventional polling method in that the CPU will not waste cycles checking the value of the local semaphore, and will also “wake up” from the low power state exactly when the interrupt arrives (and not after a random period).

Barrier Synchronization is another common synchronization technique adopted by embedded applications requiring computation running in phases. Barriers ensure that no thread/CPU starts the next phase until all have finished the current phase. By slightly extending the functionality of the original semaphore model, it is possible to have an efficient hardware support for barriers. The basic idea behind our extension is simple: after having initialized the value of the local semaphore to a negative integer (equal in absolute value

to the amount of cores present in the system), each CPU will increment by one all the other processors’ semaphores, and will wait until its local semaphore reaches a strictly positive value. As before, the CPU remains in low-power mode while the semaphore’s value is less than or equal to zero. Compared to a conventional lock-based barrier implementation, this method reduces CPU energy and bus occupancy since it avoids spinning on locks.

4. EXPERIMENTAL RESULTS

This section presents the results of evaluating the energy and performance of different synchronization mechanisms in the MPARM simulation framework. We include a varied set of benchmarks that represent applications that use shared-memory synchronization, as well as applications that use pipelines and barriers. The first group of benchmarks may benefit most from transactional memory, whereas the second may benefit from conditional synchronization.

4.1 Evaluating Embedded TM

We start with three benchmarks that require shared-memory synchronization. These benchmarks are representative of embedded applications that perform data exchanges between data structures in shared memory, such as matrices, linked lists, and red-black trees. Linked lists and red-black trees are used in memory management. Image-processing applications commonly found in plotters, printers, and digital cameras require synchronization to allow data to be exchanged within a matrix.

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 workload for the skip-list benchmark 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. Figure 3 shows the energy delay product of the skip-list benchmark. The total system energy includes the energy used by processor cores, transactional caches, L1 caches, scratchpad memories, RAM, and interconnect buses. The graph shows the effect of increasing the proportion of time spent inside a critical section when different number of cores are available. The left part of the graph (*None*) shows the default configuration, without shutting down the transactional cache, and the right part of the graph (*Agg*) shows the aggressive transactional cache shutdown policy (always shut down the TC when a transaction completes). As either the number of cores or the proportion of time spent in the critical section increases, transactional memory has an increasing energy 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. 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.

As expected, the transactional memory system offers little to no performance improvement for benchmarks that spend the majority of their execution outside of critical sections. With so little time spent performing shared memory operations, the choice of synchro-

Figure 3: Energy-delay product of the skip-list benchmark using transactions, normalized to an equivalent system using locks. Negative percentage indicate energy-delay *improvement* relative to locks.

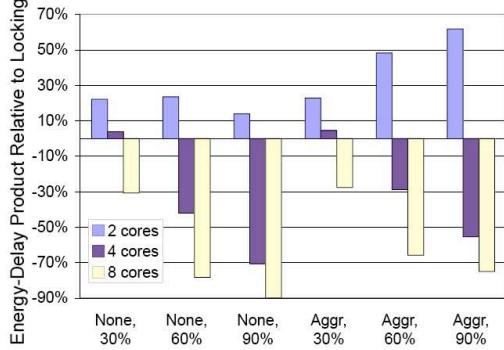
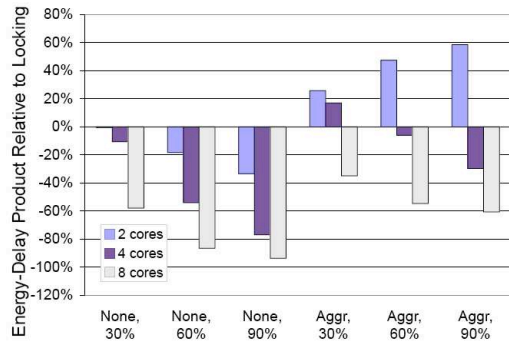


Figure 4: Red-black tree normalized energy-delay product



nization mechanism has little effect on throughput. In addition, with few cores in use, one cannot exploit transactional memory’s potential for increased parallelism. When the proportion of execution inside the critical section is increased to 60% and 90%, 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.

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. Figure 3 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. The graph also shows the scalability of transactional memory, as the energy-delay product is improved with increasing number of cores.

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 4 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 aggressive 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 (up to 50 times), resulting in a large time delay and only a small increase in energy efficiency.

Figure 5: Matrix micro-benchmark normalized energy-delay product

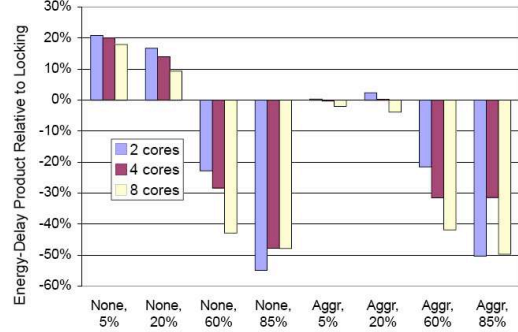
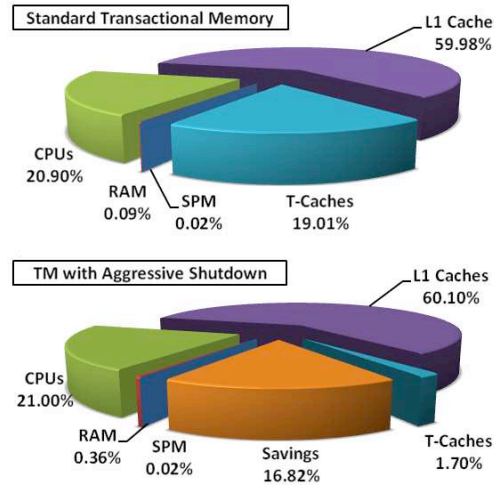


Figure 6: Energy distribution for the matrix micro-benchmark with 5% of execution inside a critical section, with 4 processors, both with and without an aggressive transactional cache shutdown policy.



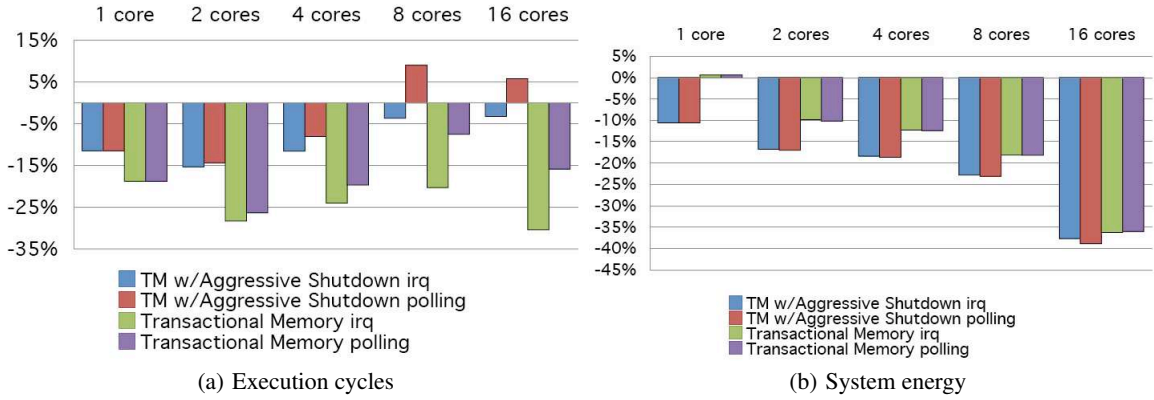
The matrix benchmark models accesses to a shared matrix. It is parameterizable, allowing us to vary the percentage of execution spent in the critical section. The difference between the micro-benchmark configurations with 5% and 20% of total execution spent in a critical section is minor, both in terms of contribution to delay and energy consumption. Both result in increased energy consumption in exchange for a small improvement in performance when using transactions. As with the previous benchmarks, but more dramatically in this case, the advantage of transactions increases with larger portion of the execution being spent in a critical section.

In contrast to other benchmarks, varying the number of cores has little impact on performance and energy consumption of the matrix micro-benchmark. This behavior is due to the structure of the micro-benchmark, which keeps the number of operations constant regardless of the number of cores. Thus, a similar workload is divided among more cores, whereas the other benchmarks increase the workload with increasing number of cores.

Due in part to lower bus traffic, aggressive shutdown does very well with the matrix micro-benchmark compared to the red-black tree and list-based benchmarks. In fact, there is a negligible difference in execution time between having an aggressive shutdown policy or none at all. For experiments with small critical sections in particular, aggressive shutdown also leads to substantial energy savings. The energy-delay results for these benchmarks are summarized in Figure 5.

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. Fig-

Figure 7: Execution cycles and system energy of the *asm-matrixdep* benchmark using transactions with and without aggressive shutdown policy, and with the *polling* or *irq* semaphores policy enabled, normalized to the locking scheme. Negative percentages indicate performance/energy advantage.



ure 6, 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 policy for shutting down the TC is enabled under the same conditions (the bottom chart of Figure 6), 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.

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.

4.2 Pipelined Applications

We now evaluate benchmarks that use pipelined synchronization. These include a micro-benchmark (*asm-matrixdep*), two embedded applications (*jpg* and *gsm*), and the *asm-fft* benchmark.

The *asm-matrixdep* benchmark implements a pipelined matrix multiplication. The application is not parallelized in the sense that adding more cores translates into a deeper computation pipeline (that is, the input data is computed multiple times according to the number of cores), and does not result in a subdivision of the workload among the processors. During the execution, each CPU acts both as a consumer and as a producer. In the consumer portion, the core verifies the availability of new data by testing its own consumer semaphore. If the semaphore is non-zero, then the CPU brings into private memory a portion of shared data, and computes the required multiplications. Otherwise, if the semaphore is zero, it aborts the transaction. Note that the check-and-abort operation is atomic. The rollback and backoff mechanisms are handled differently depending on the system policy (*i.e.*, polling vs. interrupts, or our own *irq* interrupt policy). Similarly, in the producer portion, the CPU checks its own producer semaphore and if non-zero writes back the results to shared memory. Otherwise, its transaction is aborted. If the TC is not present, then the synchronization mechanism relies on locking.

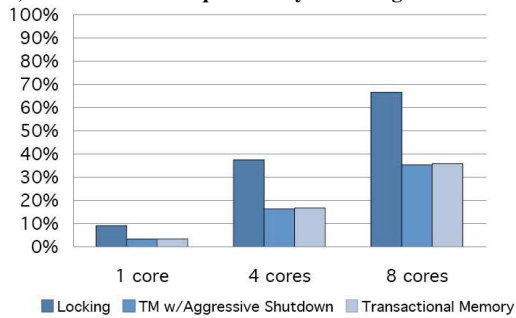
Figure 7(a) shows the execution time (relative to locking) for the TC and TC with aggressive shutdown schemes with varying number of cores and policies (*irq* vs. *polling*). Overall, transactional memory usually performs better than locking. For example, in a 16-core configuration with the *irq* policy enabled, the TC system achieves about a 30% improvement in performance. As expected, enabling aggressive shutdown generally leads to a degradation in performance compared to leaving the TC powered on, especially for systems with high numbers of cores. The crossover point is reached with the 4 CPU configuration; with more than 4 cores the TC with aggressive shutdown scheme performs slightly worse than locking. However, this price may still be reasonable if energy consumption can be substantially reduced, as discussed below.

Figure 7(b) shows the total system energy (relative to locking) for the TC and TC with aggressive shutdown architectures with varying number of cores and policies (*irq* vs. *polling*). Again, the transactional cache introduces improvements in the system (up to 35% energy savings in a 16 core configuration). However, now the aggressive shutdown policy turns out to be extremely competitive, resulting in the largest energy savings. The experimental results suggest that, in general, the TC with no aggressive shutdown is the optimal choice. However, for energy constrained systems, the results suggest that aggressive shutdown still may be considered as a valid alternative.

We also tested the architecture with two media applications: *gsm* and *jpg*. For *gsm*, three cores perform a fixed amount of lossy sound compressions and decompressions using the *gsm* library. The decoder is mapped to core 1 and core 3, while the encoder runs on core 1 and core 2. The *jpg* program implements pipelined jpeg decoding, and is parallelized for 4 cores. We replaced the critical sections with transactions and added support for the distributed semaphores. In the original versions of *gsm* and *jpg*, the access patterns in shared memory were optimized for limiting data conflicts as much as possible. The inter-processor communication relies on writes and reads in portions of shared memory that are managed as FIFO queues. The synchronization is limited to simple atomic operations on the queues’ indices.

Our results show that the execution times for both applications are not affected by the presence of a transactional cache itself. This is somewhat expected, since synchronization plays a small role here, if compared to the computational tasks. Instead, the system energy is reduced by use of the *irq* semaphore. Both benchmarks show a reduction of up to 25% in the system energy when using the TC with aggressive shutdown configuration in comparison to the locking scheme. Recall that, when using the proposed mech-

Figure 8: Bus occupancy for *asm-fft* using transactions, with and without aggressive shutdown policy, and with the *irq* semaphores policy enabled, normalized to an equivalent system using locks.



anism, each core switches its status to low-power mode whenever the FIFO queues are empty or full. In the original locking version of *gsm* and *jpg*, the processors waste energy staying in full power mode performing a busy wait on the locks instead. Since the TC is rarely used, shutting down the TC becomes an attractive solution. The energy results can be explained by considering the bus occupancy, as the use of distributed semaphores frees the bus from energy-wasting spin-lock operations.

Finally, *asm-fft*, an application performing a Fast Fourier Transform, was used to test the hardware implementation for barriers. When running *asm-fft* the CPUs spend only a small fraction of the execution time performing synchronization tasks, since the workload is dominated by the computation. We replaced the original barriers with new function calls that make use of the underlying distributed semaphores architecture. Figure 8 shows bus occupancy for the TC and TC with aggressive shutdown configurations, with the *irq* interrupt policy enabled. As before, the system benefits from the proposed architecture. In fact, even in a low-synchronization scenario, the execution time is reduced by up to 18% in an 8 core system. The bus occupancy data helps to understand the reason: when using the distributed semaphores, the system bus is freed of all the lock-spinning operations and, consequently, fewer cycles are required for the processors to access the external data. As we expected, Shutting down the TC is the optimal solution, since in this particular application there is no need for transactional memory support.

5. CONCLUSIONS

In this paper, we have described two hardware implementation techniques for obtaining energy-efficient synchronization for embedded systems: transactional memory and distributed semaphores. Our results show that because embedded systems are constrained very differently compared to general purpose systems, implementing these techniques using established mechanisms will not necessarily lead to an energy-efficient solution, and may even be counter-productive in terms of performance. To address this issue for transactional memory, 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. However, we have also shown that shutting down the TC can be counter-productive in cases where it leads to high bus traffic, so it is necessary to enable this “aggressive shutdown” policy adaptively according to application characteristics. For our implementation of distributed semaphores to support conditional synchronization, we found that in most cases workloads using these distributed semaphores (sometimes in conjunction with transactional memory) were substantially more energy-

efficient than the same workloads using locking. Much of this benefit is due to a reduced load on the system bus.

Finally, for future work, we would 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 in the future.

6. 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*, 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 *Symposium on Hardware/Software Codesign*, pages 73–78, 2002.
- [5] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA*, 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), Jul. 2005.
- [8] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). *ACM SIGOPS Oper. Syst. Rev.*, 38(5):1–13, 2004.
- [9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.
- [10] J. Li, J. Martínez, and M. Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In *HPCA*, February 2004.
- [11] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a MPSoC environment. In *DATE*, 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] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *DATE*, April 2006.
- [14] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA*, February 2006.
- [15] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *ISLPED*, August 2005.
- [16] Philips nexperia platform. www.semiconductors.philips.com.
- [17] PC205 platform. www.picochip.com.
- [18] F. Poletti, A. Poggiali, D. Bertozzi, L. Benini, P. Marchal, M. Loghi, and M. Poncino. Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support. *IEEE Trans. Comput.*, 56(5):606–621, 2007.
- [19] Nomadik platform. www.st.com.
- [20] STMicroelectronics. www.stm.com.
- [21] OMAP5910 platform. www.ti.com.