Embedded-TM: Energy and Complexity-Effective Hardware Transactional Memory for Embedded Multicore Systems

Cesare Ferri\textsuperscript{a}, Samantha Wood\textsuperscript{b}, Tali Moreshet\textsuperscript{c}, R. Iris Bahar\textsuperscript{a}, Maurice Herlihy\textsuperscript{d}

\textsuperscript{a}Division of Engineering, Brown University, Providence, RI 02912
\textsuperscript{b}Computer Science Department, Bryn Mawr College, Bryn Mawr, PA 19010
\textsuperscript{c}Engineering Department, Swarthmore College, Swarthmore, PA 19081
\textsuperscript{d}Computer Science Department, Brown University, Providence, RI 02912

Abstract

We investigate how transactional memory can be adapted for embedded systems. We consider energy consumption and complexity to be driving concerns in the design of these systems and therefore adapt simple hardware transactional memory (HTM) schemes in our architectural design. We propose several different cache structures and contention management schemes to support HTM and evaluate them in terms of energy, performance, and complexity. We find that ignoring energy considerations can lead to poor design choices, particularly for resource constrained embedded platforms. We conclude that with the right balance of energy-efficiency and simplicity, HTM will become an attractive choice for future embedded system designs.

Key words: Hardware transactional memory, Energy-efficient design, Cache design, Contention management

1. Introduction

High-end embedded systems are increasingly coming to resemble their general-purpose counterparts. Embedded systems such as smart phones, game consoles, “net-tops”, GPS-enabled automotive systems, and home entertainment centers are becoming ubiquitous. In the same way that smart phones are gradually usurping many of the functions of laptops, specialized high-end embedded systems will eventually displace many general-purpose systems. Eventually, such devices will affect every aspect of modern life, and their energy consumption profiles will have a broad economic impact.

Like their general-purpose counterparts, and for many of the same energy-related reasons, embedded systems are turning to multicore architectures. This switch has profound implications for software, which must now manage concurrent activities. It is well-established that traditional synchronization mechanisms such as locks have substantial drawbacks. Transactional memory \cite{4} has emerged as a promising alternative.

Here, we investigate how transactional memory can be adapted for embedded systems. We then describe different designs for our Embedded-TM architecture. Transactional memory for embedded systems makes different demands than transactional memory for general-purpose systems. The principal difference is the central importance of energy consumption: although embedded systems are becoming more sophisticated, they are and will continue to be energy-constrained, either because they run on batteries, or simply because energy consumption is increasingly a concern for systems at all levels. We are the first to use energy consumption as a guide for designing transactional memory mechanisms. While most earlier work on transactional memory has neglected the question of energy consumption, we claim that it should be one of the driving concerns in transactional memory design. Taking energy into account requires revisiting and revising many widely-accepted assumptions, as well as widely-accepted architectures.

Because embedded systems are energy-constrained, there is an overriding need for simplicity: many techniques suited for general-purpose systems, such as out-of-order instruction or hardware multithreading, are too complex and power-hungry for today’s embedded systems. Any realistic transactional memory design for embedded systems must make do by combining simple components. We are willing to propose minor changes to existing standards, but not (what we consider) radical changes.

The need for energy-efficiency and simplicity makes software transactional memory (STM) unattractive. (Klein et al. \cite{5} provide an analysis of the energy costs of a typical STM system.) For most embedded applications, it is unacceptable, both in terms of performance and energy consumption, to place a software “barrier” at each memory access. Indeed, embedded applications often run without an operating system. By contrast, we will see that a simple hardware transactional memory (HTM) can both enhance performance and conserve energy.

While hardware transactional memory makes fewer resource demands than software transactional memory, limitations on cache size and associativity bound transactions’ sizes and durations. While proposals exist for “unbounded” transactional memory \cite{6, 7} that allow transactions to survive certain kinds of resource exhaustion, these schemes are much too complex to be considered for embedded systems. For most embedded sys-

\[ \text{February 4, 2010} \]
tems, however, applications’ resource requirements are well-understood, and transactions that exceed those expectations are likely to be rare. Nevertheless, it is important to understand how to structure caches for HTM in embedded systems to maximize transaction sizes without compromising performance or increasing energy consumption. We will describe several such designs.

We evaluate HTM designs using three criteria: energy, performance, and complexity. Sometimes these criteria reinforce one another, and sometimes not. Here, we investigate a sequence of HTM designs, starting from a simple baseline, and moving on to a sequence of redesigns, each intended to address a specific problem limiting energy efficiency and performance. Structuring the presentation of Embedded-TM as a sequence of redesigns makes it possible to quantify the contribution of each incremental improvement.

We take as the baseline Embedded-TM an HTM based on a simple cache architecture [2, 4] in which non-transactional data is stored in a large L1 cache, and a smaller, fully-associative transactional cache stores the data accessed within a transaction. The principal drawback of this architecture is that the transactional cache consumes too much energy. As a first line of defense, we consider how to conserve energy by powering down the cache without adversely affecting performance.

Another drawback of the baseline Embedded-TM architecture is the limited size of the transactional cache. Any transaction whose data set cannot fit in that cache cannot complete, and must continue in a less-efficient serial mode described below. To alleviate this problem, we consider an alternative design in which both transactional and non-transactional data are kept together in the L1 cache. The L1 cache is substantially larger than the transactional cache, and eliminates the need to maintain coherence across two same-level caches.

While this design supports larger transactions, it is still limited by resource constraints. To keep energy consumption down, the L1 must have limited associativity, so a transaction unlucky enough to overflow a cache line must run in serial mode. We address this problem by introducing a small victim cache to catch transactional entries evicted from the main cache [3]. Although we are back to a two-cache architecture, the victim cache is needed only when the main cache overflows, so it can be small, and powered down for longer durations.

As often happens, alleviating one problem exposes another. Transactions can also be prevented from making progress by data conflicts that occur when two transactions access the same memory location, and at least one access is a write. The first approach we consider, called eager conflict resolution, works well when transactions have few data conflicts, but less well when transactions have many data conflicts.

We examine two approaches to the problem of high-conflict transactions: a brute-force approach where a transaction that fails to make progress is eventually restarted in serial mode, and a more complicated approach where conflicts are resolved in a “lazy” manner. Lazy conflict resolution postpones the decision on aborting transactions to a later time, when more data on detected conflicts is available, thus potentially increasing concurrency.

Each approach is successful in some circumstances. The brute-force approach is attractive for its simplicity and wide range of effectiveness, and it works moderately well most of the time. The lazy mode algorithm incurs a higher overhead cost, sometimes penalizing low-conflict transactions, but is effective for high-conflict transactions.

We use a cycle-accurate simulator to investigate how well each of these Embedded-TM designs works on a range of benchmarks, as well as how simple TM designs compare to locking. Confirming prior observations [2], we find that even simple TM designs outperform locking with respect to both energy and performance. Each of the successive designs we consider improves the energy-performance product of most benchmarks. This improvement is workload-dependent in the sense that it is possible to find some configuration of some benchmark where any particular design does not improve on its predecessor, but overall each successive redesign is an improvement.

These results confirm that ignoring energy considerations can lead to poor design choices, particularly for resource-constrained embedded platforms. As architectures progress, and the demands of embedded platforms evolve, the further design and evaluation of energy-efficient cache architectures for HTM remains a promising direction for further work.

2. Architecture

We developed and tested our Embedded-TM designs using the MPARM simulation framework [8, 9], a cycle-accurate, multi-processor simulator written in SystemC. MPARM models any simple instruction set architecture with a complex memory hierarchy (supporting, for example, caches, scratch pad memories, and multiple types of interconnects). MPARM also includes cycle-accurate power models for many of its simulated devices. The power models reflect a 0.13μm technology provided by STMicroelectronics [10], and the energy model for the fully associative caches is based on Efthymiou and Gar-side [11].
Figure 1 shows the basic system configuration. It consists of a variable number of ARM7 cores (each with a 4KB 4-way L1 data cache and 4KB 4-way L1 instruction 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 [12]. A cache-coherence protocol (MESI) is also provided by snoop devices connected to the master ports. Platforms featuring such cache-coherency subsystems are common (e.g., the ARM11 MPCore Multiprocessor [13]). While the private and shared memories are sized arbitrarily large (256KB each), they do not significantly impact the performance or power of our system (see Section 3).

Each transaction is speculatively executed by the CPU. If no conflicts with another transaction are detected, its effects become permanent, and the transaction commits. Otherwise, if conflicts are detected, its effects are discarded, and the transaction aborts and is later restarted.

What does it mean to detect a conflict with another transaction? Like most HTM schemes, our Embedded-TM architecture relies on an underlying cache coherence protocol to detect when data cached by one CPU is invalidated by another (we use a conventional MESI protocol). A transaction can cache a data item either for reading or for writing. A conflict occurs when two transactions concurrently attempt to cache the same data item, and at least one caches that item for writing.

Cache coherence protocols are notoriously complicated and difficult to prove correct. Our goal of simplicity means that we consider only designs that make few, preferably not any, changes to the native cache coherence protocol. While it is easy to propose radical protocol changes on paper, our reliance on a cycle-accurate simulator “keeps us honest” in the sense that we cannot ignore implementation difficulties. We are also concerned that radical changes to highly-optimized protocols limit portability and broader acceptance.

Hardware support for transactional memory is organized around these components:

1. A cache coherence protocol for detecting data conflicts,
2. A buffer for storing and modifying transactional data, and
3. A rollback mechanism for re-executing aborted transactions.

What happens when a transaction overflows its buffer? Given our need for simplicity, we take a brute-force approach. The transaction continues in serial mode: all other CPUs are suspended, and the privileged transaction runs in isolation. It uses the entire memory hierarchy, and runs non-speculatively to completion.

Figure 2 shows an overview of our Embedded-TM architecture. To start a transaction, the CPU creates a local checkpoint by saving the register contents to a small (128B) scratch pad Memory (SPM) [14]. (The pipeline is stalled while the CPU writes to the SPM.) The SPM must be large enough to hold all the CPU registers.

2.1. The Transactional Cache

The Transactional Cache (TC) is a small (512B) fully-associative memory. Each transaction keeps two copies of the data it accesses in the TC: a working copy and a backup copy. If the data is found in the L1 cache, it must be invalidated there before being placed in the TC. The transaction modifies the working copy. If there is no data conflict, the transaction completes successfully, backup copies are invalidated, and the working copies become visible. On a data conflict, the snoop device notifies the CPU, invalidating the working copies, and restoring the backup copies. The CPU enters a low-power mode, and after a random backoff, re-executes the transaction. Starvation is practically avoided by adopting an exponential distribution for the backoff periods, and also by setting a threshold for the maximum number of consecutive aborts the system allows before switching the execution in serial mode (see Section 2.3). Note that our model includes a realistic state-switching overhead (i.e., idle to active).

When reading or writing data from memory, the TC is checked first (1 cycle). On a TC miss, the rest of the memory hierarchy (starting with the L1 cache) is searched. To save power, the caches are accessed sequentially. Since TC misses are rare (we found the average TC miss rate to be about 5%), this serial access has a negligible impact on performance.

Because the TC is fully associative, it makes effective use of limited space, but it is expensive in terms of energy and power. As shown in Figure 3, the TC accounts for about 30% of total energy consumption when running a sample benchmark.\(^1\) Making the TC set-associative would save power, but makes it more

---

\(^1\) Figure 3 shows energy distribution for the matrix-microbenchmark with 5% of the time spent inside transactions, as described in Section 3.1.
likely that a transaction may overflow a set. Restarting such a transaction in the TC would probably be futile, since the same overflow is likely to recur, forcing the overflowing transaction to run in serial mode.

It is not strictly necessary to keep valid data in the TC after a transaction commits. Earlier work [2] shows that it is often more energy-efficient to write back the modified lines to the traditional cache hierarchy after the commit, allowing the TC to be powered down when not in use. This approach is called aggressive shutdown mode. The CPU is stalled while the TC is flushed. Powering up the TC at the start of each new transaction incurs a 0.2μs (40 cycles) overhead. Powering down the TC improves energy efficiency most of the time, although it is not a good strategy when there are back-to-back transactions, since it results in unnecessary traffic between the TC and the rest of the memory hierarchy. We evaluate these issues in more detail in Section 3.3.

2.2. Size Limitations

The baseline Embedded-TM architecture with aggressive shutdown is simple, and it provides good performance and energy efficiency for transactions with data footprints small enough to fit in the TC. If the transaction’s data footprint is too large to fit in the TC, then the transaction must be run in serial mode, which is unattractive because there is no concurrency. The absence of concurrency is not troubling for transactions that have data conflicts, because such transactions must be serialized in any case. Lack of concurrency does matter for large, non-conflicting concurrent transactions. We face a dilemma: a larger TC means larger transactions, but substantially increases power consumption.

It seems wasteful to have transactions overflow the small TC when the L1 cache, normally much larger than the TC, may have plenty of room. Here is an alternative design that uses the L1 cache for both transactional and non-transactional data. Because there is much more memory to hold transactional data, the likelihood of overflow should be substantially reduced. Unfortunately, because it is impractical to make the L1 cache highly associative (especially in a power-constrained embedded platform), we have replaced one problem with another: a set-associative combined cache introduces the danger that an unlucky transaction may overflow a set while the rest of the cache still has room.

Our solution is to place a victim cache (VC) between the L1 cache and main memory to catch transactional data evicted from the L1. The notion of a victim cache is not new (e.g., [15, 16, 17]), but our application of this technique is novel because we are the first to use this notion to improve the energy and performance of HTM.

Normally, all transactional data resides in the L1 cache. The victim cache is used only when a transactional entry is evicted from the L1 cache. Here, too, transactions access the caches sequentially, testing first the L1, and only after an L1 miss testing the VC. This strategy saves energy without hampering performance since most accesses hit in the L1 cache. Because the VC is used only when a set in the L1 overflows, the VC can be designed to be smaller than the TC used in the baseline design.

Transactions execute concurrently while the VC is in use. If, despite everything, the VC overflows, then the transaction continues in serial mode. Because the combination of the L1 and VC provides much more room than the baseline TC, overflows are much less common, improving both performance and energy consumption. In terms of high-level architecture, the core design remains virtually identical to the baseline design (Figure 2), except that the TC has been replaced by a smaller VC.

Unlike the TC, the L1 and VC caches do not hold backup copies of transactional data. When a transaction aborts, the CPU refills the line from memory, which increases bus traffic and reduces both performance and energy efficiency. However, this cost should be negligible if the abort rate is low enough. Because the VC is utilized only when transactions overflow the L1 cache, it makes sense to keep the VC powered down unless needed. Like the TC, powering up the VC costs on the order of tens of cycles (i.e., 40 cycles). The experimental results in Section 3.4 show that, overall, the L1+VC design is better than the baseline TC design, both in terms of energy and performance.

2.3. Reducing Abort Rates

Reducing transaction overflow exposes more parallelism, which in turn exposes more data conflicts. All Embedded-TM designs described so far use eager conflict detection and resolution, meaning that conflicts are detected and resolved when a transaction accesses a location, rather than waiting until a transaction is ready to commit. Eager detection is attractive because it requires minimal modifications to the original cache coherence protocol. In particular, neither new bus states nor new coherence signals are needed. Suppose a transaction at one CPU is about to write to an address whose data has been modified by another transaction at another CPU. As part of the cache coherence protocol, the writer broadcasts an invalidate signal for that address on the bus. The other snoop devices monitor the bus, and detect the data conflict. The snoop device for the CPU whose cache holds that data informs the CPU, which aborts and restarts the transaction, typically after a backoff period. This approach is lightweight, and fits quite well with the hardware restrictions of an embedded platform. A design that relies on limited hardware modifications not only eases verification, but also significantly increases the portability of such methods to other invalidation-based cache coherence schemes (e.g., MOESI, MSI).

Nevertheless, applications with high data conflict rates tend to perform poorly under an eager scheme. For example, long-running transactions could be repeatedly aborted by shorter conflicting transactions [18], yielding high abort rates, lower performance, and higher energy consumption. We next consider two alternative conflict resolution schemes. The first is simply to revert to serial mode (forced-serial) as soon as the abort rate exceeds a threshold [19]. This approach is attractive for its simplicity: only a few simple abort counters and control signals need to be added to the existing design, since hardware is already in place to handle serialized execution in case of an overflow.

The second, more complex alternative is a lazy conflict resolution scheme. As before, data conflicts are detected as they
occur, but instead of resolving them eagerly when they are de-
tected, they are (lazily) left unresolved until commit time. This
scheme requires more substantial modifications to our embed-
red platform, but understanding the effectiveness of such trade-
offs is essential to understand how to adapt HTM to embedded
platforms.

Figure 4: Architectural modifications needed to support “lazy abort”.

We are not the first to consider hardware support for lazy con-
flict resolution. Shriraman et al. [20] propose a hardware lazy
conflict resolution method with the goal of separating version
management and conflict resolution mechanisms in an HTM. Our
contribution is to explore the energy implications of such
an approach for mechanisms appropriate for embedded sys-
tems.

Here is how the lazy scheme works. Each CPU keeps two \( N \)
-bit bitmaps, where \( N \) is the number of CPUs. The \( \text{RW} \) bitmap
tracks which CPUs wrote data that the local CPU had previ-
ously read (see Figure 4 for an architectural overview). The
\( \text{WR} \) bitmap tracks which CPUs read data that the local CPU
had previously written. Both bitmaps track which CPUs wrotedata that the local CPU had previously written. See Figure 5
for a flowchart depicting how bits are set in the MS and MS
bitmaps.

Figure 5: Flowchart for setting bitmaps to support lazy conflict resolution.

Our lazy scheme effectively serializes transaction commits,
somewhat like TCC [21]. When a transaction is ready to com-
mitt, it asserts a signal to notify the other CPUs that a commit
protocol has started. When the other CPUs detect this signal,
they suspend what they are doing to participate in the protocol.
When all CPUs have paused, the committing CPU broadcasts
its WR bitmap and ID on the bus. Each CPU currently running
a transaction checks whether its ID is in the committing transaction’s WR bitmap, or whether the committing ID is in its local
RW bitmap. If so, that CPU aborts, invalidates cached trans-
actional entries, and broadcasts its ID. All transactions remove
the IDs of the committing and aborting transactions from their
local bitmaps. The commit protocol ends when all transactions
have been heard from.

We would have preferred to use a non-blocking lazy protocol
that did not serialize commits. Unfortunately, we found that
any such protocol would require pervasive changes both to the
underlying hardware (for example: extra bus wires, more kinds
of interrupts, etc.), and to the cache coherence protocol (there
are many subtle race conditions). Such changes would take us
too far from mainstream embedded system architectures.

We found that lazy conflict resolution improves both the per-
formance and energy efficiency of the high-conflict benchmarks
that fared poorly under eager conflict resolution. However, al-
though the serialization overhead of lazy conflict resolution is
low, it tends to penalize low-conflict applications. These results
are described in detail in the next section.

3. Experimental Results

In this section we evaluate our proposed Embedded-TM de-
design using a mix of applications. We first describe the bench-
marks used in our experiments as well as our experimental
setup, followed by a detailed discussion of our results.

3.1. Software

To test our ideas, we chose a range of different applications.
Four of these applications were taken from the STAMP bench-
mark suite [22] and one from the MiBench suite [23]. Note
that we selected a representative subset of the STAMP bench-
mark suite. Porting the entire STAMP suite is non-trivial due
to inherent limitations of the simulation platform, and is left for
future work.\(^2\)

\(^2\)For example, the lack of OS requires a workaround for handling I/O, and
no CAS instruction support means implementing locks via a semaphore-based
Test&Set.

- **Vacation** (STAMP): implements a non-distributed travel
  reservation system. Each thread interacts with the
database via the system’s transaction manager. The
application features large critical sections.

- **K-means** (STAMP): a partition-based program (com-
  monly found in image filtering applications). The number
  of objects to be partitioned is equally subdivided among
  the threads. Barriers and short critical sections are both
  used to provide synchronization.

- **Genome** (STAMP): a gene sequencing program. A gene is
  reconstructed by matching DNA segments of a larger gene.
The application has been parallelized through barriers and
large critical sections.
- **Labyrinth** (STAMP): the program finds the shortest-distance paths between pairs of starting and ending points in a 3D maze using Lee’s algorithm. Synchronization is provided by a mix of short and large critical sections.

- **Redblack, Skiplist**: applications operating on special data structures (i.e., redblack-trees and skip-lists). The workload is composed of a certain number of atomic operations (i.e., inserts, deletes and lookups) to be performed on these two data structures. Redblack-trees and skip-lists constitute the fundamental blocks of many memory management applications found in embedded applications.

- **Matrix Microbenchmark**: implements a filtering method on a matrix of data and is representative of image-processing applications typically found in plotters, printers, and digital cameras. The data is partitioned into overlapping sub-matrices, each mapped to a particular thread. The workload can be parameterized in terms of time spent executing critical sections. In our simulations we set the workload as 5%, 20%, 60% and 85% of the time spent within critical sections.

- **Patricia (MiBench)**: uses a Patricia Trie data structure to quickly perform IP address prefix matching for applications such as an IP subnet, or network or routing table lookups. The computation has been equally distributed among the cores, so that each thread operates on a subportion of the original input data. Note that in the lock-based version a global lock must be acquired by each thread before starting a new operation.

For each set of applications we also considered an average of the results, called the “Application Mix”.

### 3.2. Hardware

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Configuration(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARMv7, 3-stage in-order pipeline, 200MHz</td>
</tr>
<tr>
<td>L1 cache</td>
<td>4KB 4-way lcache, 4KB 4-way Dcache</td>
</tr>
<tr>
<td>Cores</td>
<td>[4, 8]</td>
</tr>
<tr>
<td>Tx Policies</td>
<td>TM-vanilla, TM-shutdown-L1WB, TM-victim</td>
</tr>
<tr>
<td>Contention Management</td>
<td>eager, lazy, forced-serialization</td>
</tr>
<tr>
<td>TC, VC</td>
<td>[Fully Associative], [64B,512B]</td>
</tr>
<tr>
<td>Bus</td>
<td>Amba AHB</td>
</tr>
</tbody>
</table>

Table 1: Overview of the system configurations.

For convenience, in Table 1 we report the principal system parameters and relative configurations. We considered the following alternative Embedded-TM implementations.

- **TM-vanilla**: the original transactional memory implementation [4, 2], consisting of an additional transactional cache (TC). The transactional data resides exclusively in the fully-associative TC. The TC is never turned off.

- **TM-shutdown-L1WB**: same as vanilla TM, except that the TC is shut down after each commit, as described in Section 2.1. Before turning off the TC, the CPU writes back the modified TC lines into the L1 cache. The overhead of turning the TC back on is 0.2µs (i.e., the CPU will stall for 40 cycles when reactivating the TC).

- **TM-victim**: the combined L1+victim cache configuration. The fully-associative VC contains the lines that were evicted from the (transactional) L1 cache because of conflict misses. Similar to TM-shutdown-L1WB, the VC is shut down at the end of each transaction. The modified VC lines will be written back into the main memory (i.e., SRAM).

In addition, we consider the following contention management schemes:

- **eager**: the system resolves a conflict as soon as it is detected (i.e., when a transaction tries to access a modified line in shared memory). The requesting transaction always wins the conflict, effectively aborting all other conflicting transactions. This is the default scheme.

- **lazy**: as with eager, detection happens when the conflicting data access occurs; however, conflicts are not resolved until a transaction completes. The first transaction to complete “wins”, and effectively aborts all other conflicting transactions.

- **forced-serial**: can be run on top of eager or lazy contention management. The system reverts to serialized execution if a transaction has been aborted more than once. Once the transaction completes, the system reverts back to its original conflict resolution policy (i.e., eager or lazy).

Finally, we also considered two different locking schemes for comparison:

- **locking**: a conventional Test&Set, or spin-locking mechanism. The spin-lock implementation consists of a repeated test on a variable in shared memory, and therefore requires an access to the bus for each access to the lock. The Matrix microbenchmark uses fine-grained locks, the rest of the benchmarks use coarse-grain locks.

- **locking-sleep**: a more energy-efficient locking scheme that tries to avoid wasted energy due to “busy-waiting” until the lock is free. At the first attempt to access a busy lock, the core switches into a low-power state and is woken up as soon as the lock is released by another core. The state-transition overhead when going between active and sleep mode is set to 2µs (40 cycles). We assume the power consumed by a “sleeping” CPU is 10% of a full-running core (e.g., [24]).

All Embedded-TM configurations incur a penalty of 2µs whenever a core wakes up from the power-idle state because of an abort due to a data conflict. This specific value was chosen.
since it is consistent with that found in real embedded systems (e.g., [25, 26]).

Note that while prior work used the TC along with a direct-mapped L1 cache, in this study we increased the associativity of the L1 to 4-way (which is a common degree for the associativity of data caches in embedded platforms, e.g., [25]). Our initial experiments showed this configuration to be the best in terms of energy-delay product for all benchmarks. Therefore, all experimental results shown in this paper assume the 4-way configuration for the L1 cache.

### 3.3. Evaluating Locking vs. HTM using a TC

For each application run, we measured both the total execution cycles and the consumed energy. Then we quantified the energy/performance trade-off by considering the Energy-Delay Product (EDP).

Figure 6 shows three graphs, reporting the cycles, energy, and EDP data for the different applications. Note that all results are reported relative to the simple locking scheme. The energy includes the energy used by processor cores, transactional caches, L1 caches, scratch pad memories and RAM. The matrix microbenchmark results show the effect of increasing the proportion of time spent inside a critical section when different number of cores are available. Experiments with longer proportion of time spent in critical sections have a larger total number of instructions executed per transaction. This figure shows that for most benchmarks the aggressive shutdown policy has a better EDP than the vanilla configuration. In some cases, such as Vacation, the overhead of turning the TC off and back on results in a notable increase in the number of execution cycles, which is more than compensated for by increased energy efficiency. Conversely, the aggressive shutdown policy is disadvantageous for the Redblack and SkipList benchmarks. These benchmarks show a negligible abort rate, and spend close to 90% of their execution inside critical sections. For these reasons, shutting down the TC between transactions does not result in overall energy savings. Note also that TM-shutdown-L1WB performs better in Patricia than in Redblack, because of the different workload configuration of the two applications. In Patricia, not only do threads spend less time executing transactions (80% vs. 90% in Redblack), but also conflict more often due to the higher number of object insertions (17% vs. 9% in Redblack).
3.4. Evaluating the TC Schemes vs. L1+VC Scheme

The next set of experiments focused on evaluating the potential benefits of using a unified L1+VC scheme. As mentioned in Section 2.2, an HTM scheme that relies on keeping all transactional data in a small, fully-associative transactional cache may be severely limited in the type of applications that can efficiently run on the embedded system. That is, applications with large transactions may overflow the TC and therefore will be forced to run serially, even if data conflicts with other transactions do not exist. The TM-victim scheme attempts to reduce the occurrence of overflow by allowing transactional data to reside primarily in the (much larger) L1 data cache. Furthermore, the VC serves as additional storage in case of a conflict miss in the L1, thereby providing additional defense against overflows.

Figure 7 compares the energy-delay product for the STAMP and Patricia applications when using a traditional TC scheme (both with and without shutdown) and an L1+VC scheme. Note that all results are reported relative to the TC with shutdown scheme (TM-shutdown-L1WB) since this was the overall winner from the results reported in the previous section. We focus on these applications since they were the ones that suffered from high overflow rates under the TC-vanilla and TC-shutdown-L1WB schemes and therefore stand to gain the most from the L1+VC scheme.

We see that, in most cases, the TM-victim configuration offers the best EDP. The only exceptions are K-means and Labyrinth. Labyrinth shows an interesting behavior. Our experiments reveal that, though TM-victim is capable of boosting the performance by more than 30% compared to TM-aggressive-L1WB, this improvement is not sufficient to compensate for the non-linear increase in energy consumption. A more detailed analysis shows that the L1 cache is responsible to a large extent for the energy overhead. This is due to the fact that now the L1 is used for both transactional and non-transactional data, and also because more instructions must be fetched from the icache and re-executed in case of abort (since TM-victim allows larger transactions to be aborted). K-means, as previously noted, exploits a barrier type of thread-synchronization. Since transactions are quite rare and small (i.e., less than 5% of time is spent within transaction), shutting down the TC is the most effective policy.

In contrast, other workloads benefit from the higher throughput guaranteed by the TM-victim solution. For example, for the Genome benchmark, we see that TM-victim has a 5X improvement in EDP compared to TM-vanilla for the 8 core 64B configuration. Using the TM-shutdown-L1WB scheme improves EDP a little relative to TM-vanilla, mainly by avoiding accesses to the TC when not executing transactional code. However, since the TM-shutdown-L1WB scheme does not address the problem of overflows, it will not be sufficient in the case of large transactions.

The general trend to note here is that when a system is executing large transactions, the victim configuration can often lead to a significantly better energy-delay product compared to a TM-vanilla configuration.

Figure 8 shows the transaction overflow rate for STAMP, Patricia and the Application Mix. Using the TM-victim scheme reduces the overflow rate to zero for all these bench-
As expected, the lazy contention policy reduces considerably the number of aborts. On average, we see that with no forced serialization the abort rate drops from 29% with eager to 9% with lazy. The overall effect of the increased throughput is an average improvement in EDP of about 14%. The forced-serial policy has a lighter impact on the abort rate, and affects mainly the eager case. For example, the EDP for the eager configurations is improved by 17% in Genome, and 13% in Patricia when the system reverts to forced serialization after only 2 aborts for a transaction.

We also see that the improvements in EDP are more conspicuous in Genome, Matrix-Microbench and Patricia. These applications are characterized by: 1) a high percentage of time spent in critical sections, and 2) a high abort rate. In contrast, the improvements are more constrained whenever the abort rate is small (i.e., Vacation), or the time spent in critical sections is low (i.e., K-means). Note that the reason for the high abort rate in K-means is that the majority of the transactions are executed right after leaving the barriers. Therefore, since the cores start the transactions simultaneously, the likelihood of conflict increases.

The Redblack benchmark exhibits an interesting result (Skiplist has been omitted since it exhibits similar characteristics, EDP and abort rates). Since there are no aborts, ideally we would expect the same EDP results, regardless of the particular contention policy. However, we see that the EDP data for the lazy scheme is about 30% worse. This is due to the overhead introduced in the commit phase, when cores must synchronize for reading/writing the conflict bitmaps. Figure 10 quantifies this overhead by comparing the the execution cycles of the proposed lazy scheme with an “ideal” version of the lazy scheme that instantaneously pauses all cores and serializes commits. The values in this Figure are relative to eager with no forced serialization. In general, we see that lazy and ideal-lazy show similar results (on average the difference is about 5%), confirming the efficiency of the proposed implementation.

4. Related Work

There are many mechanisms for synchronizing access to shared memory. Today, the two most prominent are locks and transactions. While most of the literature evaluates these proposals with respect to performance and ease of use, we focus here on a third criterion important for embedded devices: energy efficiency.

Prior work includes techniques for increasing the efficiency of lock-based synchronization for real-time embedded systems. Tumeo et al. [27] proposed new techniques for efficient lock-based synchronization in FPGA-based multiprocessor systems-on-chips (MPSoCs) for real-time applications. Lee et al. [28] improved the real-time performance of embedded Linux by monitoring the lock hold times.

Other researchers have investigated the energy implications of locks for multi-processor systems-on-a-chip. Loghi et al. [29] evaluated power-performance tradeoffs in lock implementations. Monchiero et al. [30] proposed a synchronization-operation buffer as a high-performance yet energy-efficient spin

3.5. Evaluating Conflict Resolution Schemes

As mentioned earlier, another important parameter affecting the performance of a transactional memory system is the transaction abort rate. Recall that when a transaction detects a data conflict with another transaction, one of the transactions needs to abort, causing the core executing that transaction to go into a low-power state for some random backoff period while the other transaction continues to execute. Eventually, the core executing the aborted transaction is woken up so it can attempt to re-execute that transaction. This whole process consumes extra energy and cycles, and is required in order to properly synchronize the two transactions. As with the overflow case, the overall effect on the system is to serialize the execution.

As discussed in Section 2.3, ideally a “lazy” conflict resolution scheme offers some advantages in terms of through-put, since it is capable of filtering out some false conflicts that would be impossible to detect with an “eager” approach. In general, however, the price to pay for such improvements is an increased complexity of the contention management scheme. An all-hardware approach, like ours, would require non-trivial modifications to the cache coherence protocol as well as the bus protocol.

Next, we quantify the impact of the eager and lazy policies on energy consumption, and then propose an alternative lightweight solution for workloads characterized by high-conflict rates. More precisely, we propose a “forced-serial” mode that can be run on top of eager or lazy contention management, and that is triggered whenever the number of consecutive aborts reaches a certain threshold. As we explained before, once the transaction completes, the system reverts back to its original conflict resolution policy (i.e., eager or lazy).

Figure 9 reports the abort rate and the energy-delay product (EDP) for an 8 core TM-victim configuration for different benchmarks. The light and the dark colored bars represent, respectively, the data for the eager and the lazy contention policies. On the x-axis we report the maximum number of retries that are allowed before forcing the system to re-execute the transaction in serial mode. The EDP values are relative to the eager scheme with no forced serialization (i.e., eager with ≥ retries).

As expected, the lazy contention policy reduces considerably the number of aborts. On average, we see that with no forced serialization the abort rate drops from 29% with eager to 9% with lazy. The overall effect of the increased throughput is an average improvement in EDP of about 14%. The forced-serial policy has a lighter impact on the abort rate, and affects mainly the eager case. For example, the EDP for the eager configurations is improved by 17% in Genome, and 13% in Patricia when the system reverts to forced serialization after only 2 aborts for a transaction.

We also see that the improvements in EDP are more conspicuous in Genome, Matrix-Microbench and Patricia. These applications are characterized by: 1) a high percentage of time spent in critical sections, and 2) a high abort rate. In contrast, the improvements are more constrained whenever the abort rate is small (i.e., Vacation), or the time spent in critical sections is low (i.e., K-means). Note that the reason for the high abort rate in K-means is that the majority of the transactions are executed right after leaving the barriers. Therefore, since the cores start the transactions simultaneously, the likelihood of conflict increases.

The Redblack benchmark exhibits an interesting result (Skiplist has been omitted since it exhibits similar characteristics, EDP and abort rates). Since there are no aborts, ideally we would expect the same EDP results, regardless of the particular contention policy. However, we see that the EDP data for the lazy scheme is about 30% worse. This is due to the overhead introduced in the commit phase, when cores must synchronize for reading/writing the conflict bitmaps. Figure 10 quantifies this overhead by comparing the the execution cycles of the proposed lazy scheme with an “ideal” version of the lazy scheme that instantaneously pauses all cores and serializes commits. The values in this Figure are relative to eager with no forced serialization. In general, we see that lazy and ideal-lazy show similar results (on average the difference is about 5%), confirming the efficiency of the proposed implementation.

4. Related Work

There are many mechanisms for synchronizing access to shared memory. Today, the two most prominent are locks and transactions. While most of the literature evaluates these proposals with respect to performance and ease of use, we focus here on a third criterion important for embedded devices: energy efficiency.

Prior work includes techniques for increasing the efficiency of lock-based synchronization for real-time embedded systems. Tumeo et al. [27] proposed new techniques for efficient lock-based synchronization in FPGA-based multiprocessor systems-on-chips (MPSoCs) for real-time applications. Lee et al. [28] improved the real-time performance of embedded Linux by monitoring the lock hold times.

Other researchers have investigated the energy implications of locks for multi-processor systems-on-a-chip. Loghi et al. [29] evaluated power-performance tradeoffs in lock implementations. Monchiero et al. [30] proposed a synchronization-operation buffer as a high-performance yet energy-efficient spin
lock implementation that reduces memory references and network traffic. Yu et al. introduced energy-efficient producer-consumer communication by compiler-inserted write-through store insertions to update a cached memory location before exiting a synchronization region [31].

Others have investigated lock-free synchronization for embedded systems. Cho et al. considered the benefits of lock-free synchronization for the multi-writer/multi-reader problem in embedded real-time systems [32]. Yang et al. showed how to exploit access pattern regularity in a single producer/single consumer synchronization to implement a light-weight synchronization mechanism that encodes dependence information within each memory access [33].

Transactional memory has been extensively investigated as an alternative means of synchronization in general-purpose systems. The principle 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. Transactional memory can be implemented in hardware (e.g. [4, 34, 21]), in software (e.g., [35, 36]), or via hybrid mechanisms that combine hardware and software (e.g. [37, 20]).

One of the distinctions between the various implementations of transactional memory is in the way they approach version management and conflict detection and resolution. FlexTM [20] includes hardware support for version management and conflict detection, with software-controlled conflict resolution. FlexTM supports a range of conflict resolution schemes, one of which can be selected according to application needs. In contrast to FlexTM, our implementation of lazy conflict resolution is centered around simplicity. It is a hardware-only implementation, which is less flexible and requires fewer architectural modifications.

Larus and Rajwar [38] provide an excellent survey of transactional memory. Because previous transactional memory proposals targeted general-purpose systems, they focused mainly on performance and ease of programming. Here, we target embedded systems, which are resource and energy constrained. For this reason, we focus on simple hardware transactional memory, with an emphasis on energy efficiency and simplicity.

Our prior work presented in [2] showed that hardware transactional memory can be implemented in embedded systems with minimal, energy-efficient hardware support. Our scheme, like all pure HTM systems, is limited to running transactions whose data sets fit in the hardware cache. Transactions that overflow the cache are run in a less-efficient serial mode. In this paper, we consider alternative cache architectures for HTM in embedded systems, architectures designed to reduce the likelihood of cache overflow with the additional goal of reducing overall energy consumption. Specifically, we proposed the use of the L1 cache as the primary storage space for holding transactional data (along with non-transactional data). In addition, we use a small victim cache to hold transactional data evicted from the L1 cache due to conflict misses, thereby reducing the occurrence of transactional overflows that force transactions to be serialized. While our proposed scheme has similarities to other transactional memory proposals that use a victim cache...
As presented in our prior works, our work is distinct in that prior work did not fully evaluate the impact of the victim cache itself on either energy or performance. In addition, since we are focusing on embedded platforms rather than general-purpose systems, our findings are driven to a large extent by the resource constraints existing in these embedded systems. As we introduce alternative transactional memory designs based on either hardware-supported lazy conflict resolution or forced serialization to better target workloads with highly conflicting transactions. Finally, we include extensive experimental results beyond those reported in [3] to demonstrate the advantages as well as the drawbacks of each of our designs.

Unbounded (or virtualized) transactional memory [6, 39, 34, 7] proposals include additional hardware structures to allow transactions to continue after overflowing the L1 cache, and even to migrate from processor to processor. While some of these proposals may be attractive in general-purpose systems, they are too complex for today's embedded systems.

The permissions-only cache (PO cache) of Blundell et al. [40] addresses the same problem as our victim cache: minimizing transaction overflow. On an overflow, speculative data is written back to memory, and the original values are logged in thread-local storage, but the (much smaller) permission bits are kept in the cache, allowing the cache coherence protocol to continue to detect conflicts. (If the PO cache itself overflows, then an additional serialization mechanism is called into play.)

While the PO cache scheme may be attractive for general-purpose architectures, it is incompatible with our goal of minimizing changes to the underlying embedded architecture. Maintaining the undo-log, in fact, would require not only non-trivial changes to the CPU pipeline (since a write operation should be monitored and properly propagated into the log), but also cost extra cycles even for the case of non-conflicting transactions (since logging is not a cycle-free operation). Moreover, when a transaction aborts, the PO cache scheme must restore the original memory state from the log, blocking (or perhaps restarting) any concurrent transactions that attempt to access that data while recovery is in progress. This functionality would require substantial changes to the base architecture, tracking more synchronization state, and adding new states, messages, and behaviors to the standard cache coherence protocols. These changes go far beyond those needed to support a victim cache.

5. Conclusions

Like general-purpose systems, today’s embedded systems are adopting multicore architectures. In the medium term, advances in technology will provide increased parallelism, but not increased single-thread performance. System designers and software engineers can no longer rely on increasing clock speed to enable ever more ambitious applications. Instead, they must learn to make effective use of increasing parallelism. Transactional memory is an attractive way to structure concurrent programs.

Nevertheless, transactional memory for embedded systems is different from transactional memory for general-purpose systems. The principal difference is the importance of energy efficiency. While performance remains important, we are no longer willing to consider high-performance algorithms that consume excessive amounts of energy.

We have shown that hardware transactional memory is a viable way to structure synchronization on power-constrained embedded systems, as long as it is designed with energy efficiency in mind. A straightforward implementation of Embedded-TM, using a dedicated, fully-associative transactional cache, was power-hungry, even when it could be selectively powered down. Instead, we found that we could devise a better design by mixing and matching known techniques, such as placing transactional entries in the L1 cache backed up by a small victim cache (powered down when not in use).
Surprisingly perhaps, some design decisions that individually consume more energy than their simpler alternatives yield overall energy savings. For example, in some cases employing forced serialization of transactions actually results in lower energy than attempting to maximize parallelism. We also show that a small, fully associative victim cache suffices to drastically reduce the number of overflow cases compared to a conventional HTM architecture. Given limited storage, we prefer the TM-victim scheme since it makes the most flexible use of available memory, a critical concern in resource-constrained embedded systems.

A lazy conflict resolution scheme, while adding complexity, improves the performance of high-conflict workloads, at a cost to low-conflict workloads. Perhaps the conflict resolution scheme should be configured to match the workload, either statically or dynamically.

There are still open questions. Our Embedded-TM design switches to serial mode both for transactions that overflow the hardware cache, and for transactions that repeatedly abort due to data conflicts. Further work is needed to evaluate strategies for switching aborted transactions: should one switch right away, on the grounds that data conflicts probably prevent the current transaction mix from executing concurrently, or is it more sensible to try several times, hoping that the conflicts are transient? More work is needed to exploit the observation that in many embedded systems, the transaction mix is often, but not always, known in advance, and to configure the cache and overflow policies accordingly.

References


