Lock Elision for Memcached: Power and Performance analysis on an Embedded Platform

Aditya G. Holla
Computer Science Department
Brown University
aditya_holla@brown.edu

Maurice Herlihy
Computer Science Department
Brown University
maurice_herlihy@brown.edu

Abstract

Multicore embedded systems have gained a significant importance in recent years due to the advent of complex applications which demand increased computational performance in an energy-constrained system. In order to address this, speculation-based techniques are widely under investigation. In this work, we evaluate how speculative techniques like Speculative Lock Elision(SLE) and conflict resolution schemes like Tranactional Lock Removal(TLR) can be applied to multi-core high-end embedded systems in a transparent way. In particular, our evaluation is done against a real-world application *Memcached* and present a detailed analysis of performance and energy consumption. We found that speculation was sometimes but not always advantageous for *Memcached*, and that the benefits of speculation are sensitive to critical section size, degree of lock contention, the retry policy, and the underlying hardware transactional memory's contention management policy.

1. Introduction

Speculative lock elision [22] (SLE) and *transactional lock removal* [23] (TLR) are techniques in which selected critical sections are executed speculatively by an underlying hardware transactional memory (HTM). For brevity, we refer to these schemes and their variants simply as "lock elision". Lock elision technique elides the locks dynamically and it provide better performance when there is high lock conflicts but low data conflicts inside critical section. Lock elision is a timely subject, as it is supported by Intel's new Haswell processor [14].

In this work, we evaluate how these techniques affect the energy consumption and performance of *Memcached* [11, 17] a general-purpose distributed memory caching system currently used by Youtube, Reddit, Zynga, Facebook, and others. *Memcached* uses an in-memory hash table that stores key-value pairs. When the cache is full, pairs are replaced in least recently used order. We chose *Memcached* because it is an example of a "real-world" concurrent application that must be retrofitted to support speculation. Recent work by Pohlack and Diestelhorst [21] shows that lock elision enhances performance for *Memcached* in the context of AMD's advanced synchronization facility. Our evaluation particularly focus on power consumption on embedded platforms.

We ported *Memcached* (version 1.2.5) to run on an embedded, shared-memory, multiprocessor-system-on-chip (MPSoC), replacing socket-based communication with shared-memory communication. Because our platform is embedded, we consider both energy consumption and performance as figures of merit.

We found that lock elision was sometimes but not always advantageous for *Memcached*. The benefits of speculation are sensitive to the following issues.

- Critical section size: the *set* operation (long critical section) often benefits from speculation, while the *get* operation (short critical section) does not. Thus, a composition having good amount of *set* operations will get benefit from speculation.
- False conflicts: the benefits of speculation are sometimes limited by contention for the data structures that track statistics, causing otherwise unrelated critical sections to conflict.
- Failover policy: TLR, a policy that relies entirely on the underlying HTM, provides better benefits than SLE, a policy that switches between locking and speculation.
- Contention management in the underlying HTM: a timestamp-based conflict resolution scheme performed better than a retry-count scheme.

These results suggest that retrofitting lock elision to a non-trivial application like *Memcached* can provide benefits, for both power and performance, but that realizing these benefits may require some care.

2. Platform

While most transactional memory research has focused on general-purpose platforms, ours is primarily concerned with high-end embedded systems such as smart phones, game consoles, GPS-enabled automotive systems, and home entertainment centers. In the same way that smart phones and tablets are gradually usurping many of the functions of laptops, specialized high-end embedded systems will eventually displace many general-purpose systems. Unlike traditional embedded systems, however, high-end embedded systems are subject to dynamic and unpredictable loads, and they are increasingly called upon to manage substantial resources in the form of memory, connectivity, and access to devices. Nevertheless, they will continue to be power-constrained, either because they run on batteries, or simply because energy consumption is increasingly a concern for systems at all levels.

Like their general-purpose counterparts, and for many of the same energy-related reasons, embedded systems are turning to multicore architectures. Currently, two most prominent synchronization models for shared memory are lock-based and speculation-based.

Our implementation is inspired by a previous work [22], which proposed a speculation-based, hardware technique called Speculative Lock Elision(SLE), which elides conservative locks in a transparent way. Thus, this solution achieves lock-free execution, in the absence of data conflicts and can be implemented entirely in microarchitecture, without instruction set support or modification to existing cache coherence protocol. In this work, our contribution lies in proposing similar technique in high-end embedded systems, keeping energy consumption as an important design requirement in addition to high-performance. Also, our integrated HW/SW implementation focuses on hardware simplicity which in turn contributes to energy efficiency.

While SLE, offers lock-free and concurrent execution when threads operate on disjoint set of data, but, in the presence of data conflicts, it relies on conservative lock-based synchronization. This in turn exposes the limitations of the locks. In order to overcome this, different conflict resolution schemes have been proposed. These schemes achieve lock-free execution, even in the presence of data conflicts, as long as there is enough resource to buffer the speculative data. Prior work [23], has proposed Transactional Lock Removal(TLR), which uses SLE for lock-elision, and provides time-stamp based conflict resolution scheme. In this work, we show how inclusion of such conflict resolution schemes benefit performance and energy.

We are not the first to look into speculation-based techniques in embedded systems. Ferri *et al.* [8–10] have investigated various energy-efficient, low-complexity transactional memory designs for embedded systems. These systems were evaluated using standard benchmarks such as ones from the STAMP [18], MiBench [12], and EEMBC [6] benchmark suites.

In this work, we turn our attention from standard benchmarks to a widely-used application, *Memcached*. *Memcached* is of interest because applying lock elision requires nested transactions, short transactions with both high and low contention, and complex function calls inside atomic blocks. Our preliminary analysis showed that some *Memcached* operations spend nearly 40% of their time in critical sections. Multiple data structures were protected by a single lock, suggesting the possibility of high lock conflicts but low data conflicts, promising territory for speculation.

This work makes the following contributions. We take a first step toward understanding how well speculative lock elision (and its variants) work on a real applications. Second, by focusing on an embedded platform, and evaluating power consumption as well as performance, we take a first step toward understanding how well these speculative techniques are suited to platforms where energy is as important as conventional notions of performance.

3. Background

Techniques for increasing the efficiency of locks in real-time embedded systems include Tumeo *et al.* [24], and Lee and Park [15]. Researchers that have investigated the energy implications of locks include Loghi *et al.* [16], Monchiero *et al.* [19], and Yu and Petrov [26]. Lock-free synchronization in embedded systems has been investigated by Cho *et al.* [3], and Yang and Orailoglu [25]. These papers look at non-transactional, non-speculative synchronization.

Speculative lock elision (SLE) was introduced by Ravi Rajwar et. al [22]. In SLE, processor speculates that a block delimited by an atomic read-modify-write operation (such as a test-and-set) and a subsequent store to the same location is a critical section protected by a lock. The processor then executes that block speculatively and predicts store operation on same memory location, which releases the lock. If no conflict is observed, then it successfully elides both store operation to lock and commits the transaction. Otherwise, it rollback the transaction and re-execute the block, possibly non-speculatively. Key advantage of SLE is, it can be implemented entirely in the microarchitecture, in a transparent way. Our implementation of speculative lock elision, is focused on both performance and energy improvement in the embedded space.

Transactional Lock Removal(TLR) was introduced by Ravi Rajwar et. al [23]. It is a hardware mechanism, to convert the lock-based critical sections transparently and optimistically into lock-free transactions. It uses SLE as enabling mechanism, and in presence of data-conflicts, instead of falling back to conservative locks, uses time-stamp-based conflict resolution scheme. We have implemented similar conflict resolution scheme keeping hardware simplicity and energy efficiency as a major concern.

Prior work [8, 10] describes the design of SoC-TM, an integrated hardware and software platform for transactional programming on embedded multicore systems. The work described here is based on SoC-TM, with additional modifications to support lock elision.

4. Architecture

This section describes the architecture of our underlying embedded platform and the ways it was modified to support lock elision.

4.1 SoC-TM: Baseline Architecture

This section gives an overview of the baseline hardware transactional memory (HTM) system, called SoC-TM, used for our experiments.

SoC-TM was developed on the MPARM simulation framework [1, 16], a cycle-accurate, multi-processor simulator written in SystemC. MPARM can be configured to model complex memory hierarchies, and includes a cycle-accurate power model. One important way in which this embedded architecture differs from conventional general-purpose architectures is that it does not support atomic read-modify-write instructions. Instead, the architecture provides an array of hardware locks, indexed by small integers, which can be used either as semaphores or as spin locks. The hardware locks provide the only means of locking. This restriction actually makes lock elision easier, since it is unambiguous when a core is acquiring or releasing a lock.

HTM in SoC-TM is provided by the following mechanisms.

- Each cache line is tagged with bits that indicate whether the line is transactional, and whether the data in the cache is the working copy or the backup copy.
- The CPUs share a *Bloom module* that
  - monitors transactional accesses and tracks them using per-core signatures, and
  - detects conflicts, and selects which transaction to roll back.
- The cache controller logic monitors transactional accesses and notifies the Bloom module when a transaction accesses cached data. It also detects cache overflow.
- Unlike best-effort HTMs, SoC-TM guarantees that every transaction eventually commits. As in TCC [13], if a transaction repeatedly fails to make progress (or if exceeds hardware resources) the transaction continues in *serial mode*: all other cores are suspended, and the privileged transaction runs in isolation. It uses the entire memory hierarchy, and runs non-speculatively to completion. Such events are rare.

4.2 EMBEDDED-SPEC

The SoC-TM design requires some modification to support SLE. These modifications, which we call EMBEDDED-SPEC, along with a design rationale and benchmarking results, are described in detail in a companion paper [2].

Because SLE requires coordination between locks and speculative computation, we extended the Bloom module to track lock use as well as transaction read and write set signatures. We also extended the Bloom module's contention management functions, adding policies to govern which transaction to roll back when data conflicts occur. On the software side, as described below, we modified the standard locking operations to switch to speculative execution as dictated by various policies.

In this work, we consider two distinct configurations of EMBEDDED-SPEC. The first, called EMBEDDED-LE (for *lock elision*), switches adaptively between speculative and non-speculative execution of critical sections. It corresponds roughly to Rajwar's SLE design [22]. The other, called EMBEDDED-LR (for *lock removal*), does not switch from speculative to non-speculative, relying on the underlying HTM to guarantee progress. It corresponds roughly to Rajwar's TLR design [23].

4.2.1 The Enhanced Bloom Module

As in SOC-TM, the Bloom module in EMBEDDED-LE is in charge of conflict detection and resolution. It snoops the bus to monitor the states of the hardware locks, so that conflicting speculative executions can be detected and rolled back when a core acquires one of these locks.

To keep track of which locks have been acquired or elided, it is not enough for the Bloom module to snoop on locking operation calls. In addition, it must also snoop on the responses to certain calls, and it must match up calls and responses. To this end, the Bloom module uses a state machine to track hardware lock state. The state machine is activated when an operation such as Test() or TestAndSet(), whose return value depends on the lock state, is detected on the bus. The state machine records which kind of call it observed, and when the hardware lock memory slave places the return value on the bus, the state machine acquires this value and decides the calling core's new state (either waiting, speculative, or locking mode). If the Bloom module deduces that a TestAndSet() call succeeded, then that lock's index is recorded in a register associated with that core.

In addition to extending the Bloom module to be aware of which core has which locks, we extended the Bloom module control logic to encompass different contention management policies, determining which transactions are rolled back when a conflict is detected.

- *Requester-abort*: Aborts the transaction that requested the address for which a conflict was detected.
- *All-abort*: Aborts all the transactions working speculatively on the same lock.

In either case, an aborted transaction, following the lock elision algorithm described in the next section, may then execute non-speculatively, and try to acquire the lock by calling TestAndSet(). If the snooping Bloom module detects that the acquisition succeeds, it rolls back the transactions for cores that elided that lock. If two or more cores try to acquire the lock, then one will succeed, and the others will spin.

4.2.2 Lock Elision Schemes

The next step is the software interface to the hardware-supported conflict detection and resolution. We focus on the Wait() and Signal() methods provided by locks.

- Wait($LockID$) is called to acquire a lock. If the lock is free, then it sets the lock state and returns. If the lock is busy, then the caller either sleeps for a fixed duration or spins, depending on whether the lock is treated as a semaphore or a spin-lock.
- Signal($LockID$): is called to release a lock. It clears the lock state and wakes up cores waiting to get this lock.

```
Function: wait_sle Input: LockID
while 1 do
    while Test(LockID) do
        /* lock not free, keep spinning */
    end
    if check_abort() then
        /* previous attempt aborted */
        end_transaction();
        if TestAndSet(LockID) then
            /* lock acquired, begin
            non-speculatively */
            return
        else
            /* lock is not free, continue
            loop */
        end
    else
        /* begin speculation */
        begin_transaction();
        return
    end
end
Function: signal_sle Input: LockID
if check_in_transaction() then
    /* end speculation */
    end_transaction;
else
    /* end non-speculation */
    Release(LockID);
end
```

Algorithm 1: wait_sle() and signal_sle()

```
Function: wait Input: LockID
if !check_in_transaction() then
    /* either this is first time execution
    of critical section or speculative
    mode in progress */
    wait_sle(LockID);
    /*wait_sle, either ends up
    speculatively or non-speculatively.
    Increment only if it is speculative
    mode */
    if check_in_transaction() then
        perProcessorCount++;
    end
else
    /*this must be a nested transaction*/
    if check_in_transaction() then
        perProcessorCount++;
    end
end
Function: signal Input: LockID
if !check_in_transaction() then
    /*non-speculative mode in progress*/
    signal_sle(LockID);
else
    /*decrement the nested level*/
    perProcessorCount–;
    if 0 == perProcessorCount then
        signal_sle(LockID);
    end
end
```

Algorithm 2: Wait() and Signal()

We modified Wait() and Signal() to accommodate two lock elision schemes.

base : As shown in Algorithm 1, a core calls wait_sle() to enter a critical section. It tests the lock status, and if it is not free, another core is executing non-speculatively, so the caller spins while the lock is busy. When the lock becomes free, the caller elides the lock and executes the critical section speculatively. As mentioned earlier, the enhanced Bloom module is responsible for detecting conflicts between speculative executions, and between speculative and non-speculative executions. The core calls signal_sle() at the end of the critical section. If the transaction fails to commit, control jumps to the line after the test. If a call to check_abort() reveals that the previous speculation failed, the caller calls TestAndSet() once, acquiring the lock if is free. Otherwise, it again tries to elide the lock. If the execution is non-speculative, then signal_sle() releases the lock. To avoid starvation, if a speculative execution encounters more than a threshold number of conflicts, execution reverts to serial mode.

nretries : We extend the prior scheme by keeping track of the number of times the core has encountered conflicts. If this number exceeds a threshold, the core will try to acquire the lock and execute non-speculatively.

4.2.3 Nested Transactions

Because *Memcached* uses nested critical sections, lock elision must support nested transactions. The literature [7, 20] includes three kinds of nesting: flat, closed, and open. *Flat nesting* is the simplest: all operations are executed in the outermost enclosing transaction. Aborting an inner transaction aborts higher level transactions as well. *Closed nesting* allows inner transactions to abort without necessarily aborting any enclosing transactions. *Open nesting* allows inner transactions to release side-effects, relying on compensating actions to roll back changes as necessary.

Although closed nesting provides the most useful semantics, it is too expensive to implement with limited hardware resources, so our implementation uses flat nesting. Calls to wait_sle() and signal_sle() are executed only at the top level; the rest are elided as shown in Algorithm 2.

4.3 EMBEDDED-SLEEP

In the EMBEDDED-SLEEP variation of EMBEDDED-LE, instead of spinning on a busy lock, wait_sle() switches the core to a low-power, idle mode until the lock becomes free. This scheme trades energy for latency: switching from an idle state to a standard state takes 2ms.

4.4 EMBEDDED-LR

In the EMBEDDED-LR variation, the Bloom module assigns a priority to each running transaction. When a conflict occurs, the Bloom module directs the core with the lower priority to roll back and wait until the other commits. Ties are broken by aborting the requester. This change affects only hardware, and is transparent to software.

We considered two ways to assign priorities.

- *conflict-based*: Each new transaction starts with the same priority. Each time a core loses a conflict, its priority increases by a fixed amount.
- *timestamp-based*: Each new transaction's priority is its starting time, so the longest-running transaction will have the highest priority.

5. Experimental Results

This section describes the results of applying various forms of lock elision to *Memcached*.

5.1 Memcached

Memcached is a distributed in-memory software cache, where multiple clients place key-value pairs on multiple servers. It provides operations such as get(), replace(), delete(), and compare−and−swap() on these keys. Clients can also request the current cache status, the number of keys in the cache, and related statistics. Each server dedicates a fixed amount of memory to its part of the cache, and the ensemble is viewed as a single logical cache. Clients choose a server based on key hash value. When the cache is full, pairs are replaced in least recently-used order.

Memcached creates multiple threads to handle client requests. Among these threads, one dedicated thread is responsible for distributing newly arriving TCP/UDP connections to the other threads in round-robin order. These threads listen on these client connections and respond to requests. *Memcached* has its own slab memory allocation scheme. Initially, a small number of slabs is allocated, and additional slabs are allocated as needed.

Threads within a *Memcached* server synchronize via Pthread locks. Hash table and slab memory updates are protected by the cache_lock and slabs_lock respectively. *Memcached* maintains cache statistics, such as the number of cache misses and hits. Updates to statistics are protected through the stats_lock. Other locks are used while opening and closing the connections. Our tests indicates that cache_lock and stats_lock have high contention.

5.2 Experimental Setup

In SoC-TM, as in many embedded platforms, applications run directly on the hardware, without an operating system. Adapting to this environment required substantial modifications to remove dependencies on external libraries without changing the overall design, and without changing any critical sections.

We ported *Memcached* to our embedded platform, in a way that preserves the operations and critical sections of the standard *Memcached* server. We replaced the original network message-passing communication scheme with a scheme employing shared memory, allowing some cores to act as clients while others act as server threads. We used *Memcached* version 1.2.5. In the new design, a few cores generate client requests like get() and set(), and place them in shared memory. The remaining cores act like *Memcached* servers. As in the original, one core takes the role of distributing connections to the others, each of which accepts requests, acting like a *Memcached* thread. The original critical section structure remains unchanged.

Memcached settings have been given values appropriate for the EMBEDDED-SPEC platform. The minimum chunk size is 48 bytes, the chunk factor is set to 1.25, and the maximum size of the cache is set to 60MB. Our workload generator continuously generates new connections, new key-value pairs, and requests, placing them in memory locations shared with cores executing as servers.

5.3 Memcached Critical Sections

We first compare EMBEDDED-LE, EMBEDDED-LR, as well as lock-based and HTM-based implementations. The EMBEDDED-LE configuration uses the *requester-abort* policy and *base* lock elision scheme, while the EMBEDDED-LR configuration uses the *conflict-based* policy. The HTM implementation uses "eager" conflict resolution.

The get() and set() requests are the most common, so we focus on them. We first evaluate set() in isolation. The set() operation acquires cache_lock several times to manipulate the hash table and slab memory data structures. It acquires the stats_lock to update

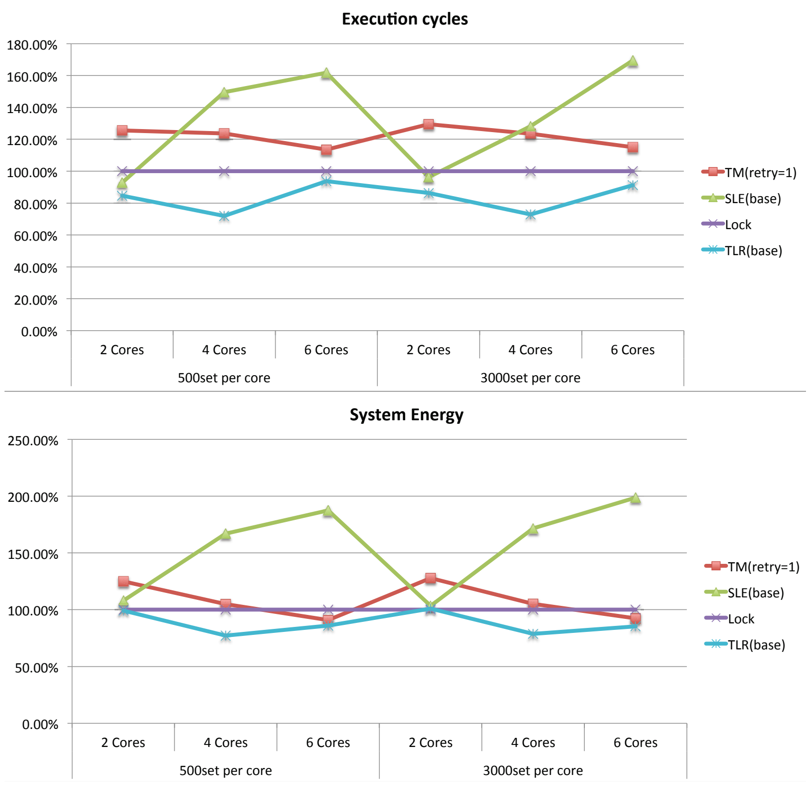

Figure 1. Execution cycles and energy consumption of *Memcached* set commands.

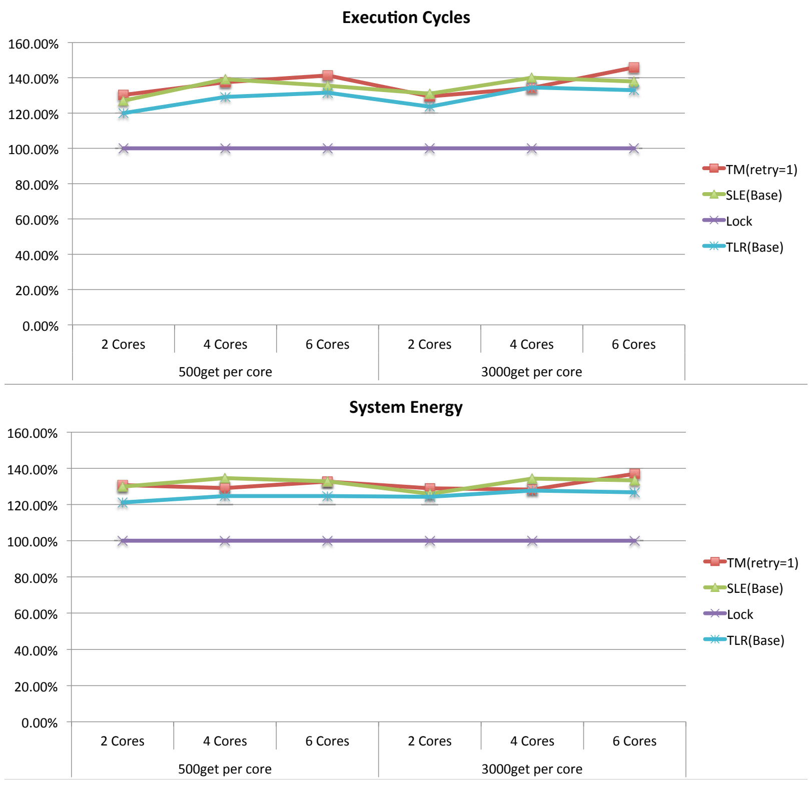

Figure 2. Execution cycles and energy consumption of *Memcached* get commands.

Number of set requests per core	Cores	TM(retry=1)	EMBEDDED-LE(base)	Lock	EMBEDDED-LR(base)
500	2	156%	100%	100%	83%
	4	129%	249%	100%	55%
	6	103%	303%	100%	80%
3000	2	165%	99%	100%	87%
	4	129%	219%	100%	57%
	6	106%	336%	100%	77%

Table 1. Energy-delay product of *Memcached* set commands. Values are relative to standard locking. Gray cells are better.

Number of get requests per core	Cores	TM(retry=1)	EMBEDDED-LE(base)	Lock	EMBEDDED-LR(base)
500	2	170%	164%	100%	145%
	4	177%	187%	100%	160%
	6	187%	179%	100%	164%
3000	2	166%	164%	100%	153%
	4	172%	188%	100%	171%
	6	199%	183%	100%	168%

Table 2. Energy-delay product of *Memcached* get commands. Values are relative to standard locking.

the statistics, and slabs_lock to update the metadata related to slab memory. While processing a single set() call, a core spends nearly 40% of its time in critical sections.

Figure 1 and Table 1 show the execution cycles, energy consumption and energy-delay product (EDP) of various speculation schemes executed with different numbers of set() requests and different numbers of cores. Graphs are normalized to the performance of standard locks.

For EMBEDDED-LE (base), both performance and energy deteriorates as the number of cores increases. This decline is mainly because aborted cores makes many speculative attempts, and most of these retries fail. This trend increases along with the number of cores.

To establish a baseline for understanding the impact of contention management, we started by replacing each critical section with a transaction executed by an *eager* HTM [10]. In this context, "eager" means that if transaction A accesses data held by transaction B, then B is always rolled back. As expected, the eager HTM implementation did poorly, about three times worse than locking . This poor performance is largely due to repeated conflicts on contended data. Restricting the number of retries and executing in serial (low power) mode, improves both performance and energy consumption. Table 1 shows that as the number of requests and cores increases, both performance and energy consumption approach that of standard locking. However, increasing the number of retries before entering serial mode decreases performance.

EMBEDDED-LR (base) performed better than locking. Both performance and energy consumption improved by about 15%, and at 6 cores the EDP improved by about 25%. This improvement is mainly due to spending more time in critical sections, and in the larger proportion of successful transactions (about 80%). Unlike the eager HTM, EMBEDDED-LR ensures that at least one contending core will go on to commit.

These observations suggest that executing critical sections of the set() command speculatively via EMBEDDED-LR can provide both performance and energy benefits. Comparing HTM and EMBEDDED-LE, repeated retrying of failed speculation is not a good idea in the presence of highly-contended locks like the stats_lock.

Lock elision did not benefit the get() command, which acquires the cache_lock and stats_lock, spending only 16% of the time in the critical section. As shown in in Figure 2 and Table 2, the EDP for EMBEDDED-LE (base) and TM (retry=1) was 80% worse, and for EMBEDDED-LR (base) 60% worse.

5.4 Evaluating Lock vs. EMBEDDED-LR

Having examined the behavior of individual get() and set() calls, we turn our attention to mixtures of these calls. Since EMBEDDED-LR (base) yielded the highest benefit for individual calls, we focus on that configuration for now. Figure 3 shows the execution cycles, energy consumption, and EDP for different ratios of get() and set() calls. The figure shows that at 60:40 ratio, EMBEDDED-LR (base) achieves about 4% improvement in execution time, and about 8% improvement in EDP. As the number of set() calls increases, both execution time and energy consumption improve. Figure 4 shows that the abort rate stays almost constant across all mixtures. Also the system spends more time executing successful transaction as the number of set() calls increases. For ratios of at least 40% set() commands, EMBEDDED-LR (base) is beneficial both in terms of performance and energy.

5.5 Other Policies

In this section, we examine other contention management policies and lock-elision schemes. We compare alternatives to EMBEDDED-LR (base), since that scheme yielded the highest benefits so far. Table 3 shows the performance and energy breakdown, as well as EDP, when tested on 6 cores with a ratio of 60:40 get() to set() calls.

Earlier we observed that EMBEDDED-LE (base), which retries speculative executions many times, does not perform well. We restricted the number of retries by using *nretries* lock-elision scheme. For a single retry, results closely match EMBEDDED-LR (base), similar to the behavior of the eager HTM with serial mode failover.

In both cases, we observed that most of the conflicts occur in the critical section protected by stats_lock, which are short but have a high conflict rate. On few occasions, slab memory metadata updates protected by slabs_lock results in data conflicts. These locks are accessed by acquiring the central cache_lock, the outermost nested lock shared by other transactions. In case of data conflicts, retrying the outer cache_lock causes false conflicts on unrelated transactions. With EMBEDDED-LE (base), we observed 60% false conflicts due to acquisition of cache_lock. Here, retrying speculation only increases the conflict rate.

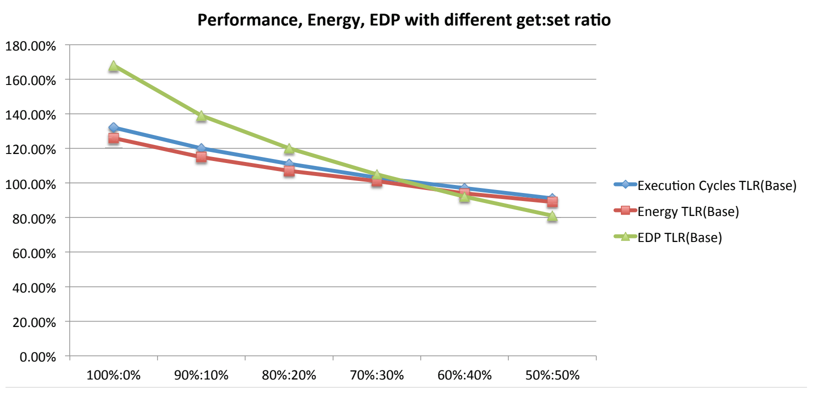

Figure 3. Execution cycles, energy, and energy-delay product of EMBEDDED-LR (base) drawn relative to locks(100%). Configuration: 6 cores, 18K requests

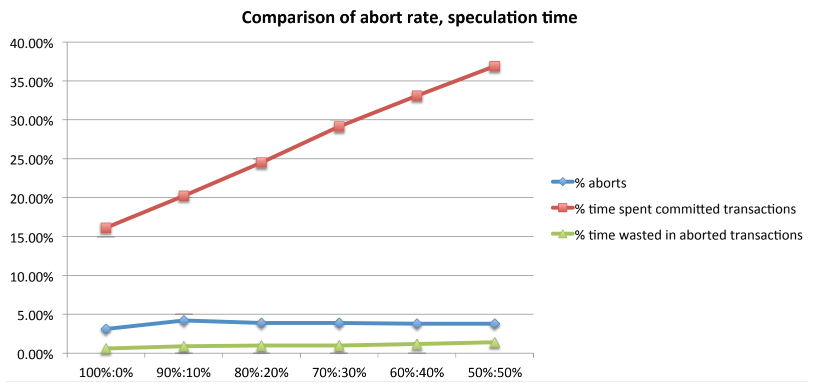

Figure 4. Abort rate, execution time of successful and aborted transactions. Configuration: 6 cores, 18K requests

Using EMBEDDED-LE with the *requester-abort* policy, whenever a core A abandons speculative execution and acquires a lock, it will cause the other cores working speculatively on that lock's critical section to abort. As a result, work may be wasted in the interval between when A aborts and when it acquires the lock. The *abort-all* policy is intended to reduce this waste by pre-emptively aborting the cores running in a critical section whose lock is about to be acquired. However, Figure 3 shows that this approach is not effective for *Memcached*.

EMBEDDED-SLEEP differs from EMBEDDED-LE (base) as follows: instead of spinning on a bus lock, it enters a low-power sleep mode. This approach improves both performance and energy consumption, yielding a EDP equivalent to that of EMBEDDED-LR (base).

The EMBEDDED-LR (timestamp) policy yields a 4% better EDP than EMBEDDED-LR (base), which is better than the other schemes, although it does require slightly more complex hardware support.

6. Conclusions

We examined the power and performance benefits of applying lock elision to *Memcached* on an embedded platform. We found that lock elision can provide non-trivial benefits to both power and performance, although not all hardware configurations were beneficial. The existence of false conflicts, especially around statistics-tracking data structures, seems to present an obstruction to further improvement.

Damron et al. [5] and Click [4] also report that false conflicts caused by shared performance counters and statistical data structures limited the performance benefits of replacing locks with speculative transactions. In our study, we were careful to preserve the locks and critical sections of the original *Memcached* when we ported it to our embedded platform. Future work have to investigate the degree to which simple refactoring of known hot-spots such as the statistics data structures can enhance the benefits of lock elision.

Acknowledgments

This project is done as part of collaborative project with Computer Science Brown University, School of Engineering Brown University and Engineering department at Swarthmore College. I'm thankful to all the members of this project and weekly group discussion, including Giuseppe Capodanno and Dimitra Papagiannopoulou. My special thanks to Prof. Maurice Herlihy, R. Iris Bahar, Tali Moreshet for their guidance and valuable feedback during submission of this work to workshop on transactional computing.

	Total requests	EMBEDDED-LE requester-abort retry=1	EMBEDDED-LE requester-abort retry=2	EMBEDDED-LE all-abort	EMBEDDED-LE-sleep requester-abort	EMBEDDED-LR abort-based	EMBEDDED-LR timestamp
	1.5K	101%	100%	105%	100%	100%	98%
	3K	94%	95%	102%	97%	100%	97%
	6K	94%	95%	104%	96%	100%	97%
Cycles	10K	96%	97%	101%	95%	100%	97%
	12K	98%	98%	101%	95%	100%	97%
	14K	96%	97%	100%	94%	100%	97%
	18K	97%	97%	104%	96%	100%	97%
	1.5K	108%	110%	120%	108%	100%	98%
	3K	108%	108%	119%	106%	100%	97%
	6K	108%	108%	121%	105%	100%	98%
Energy	10K	108%	109%	128%	104%	100%	98%
	12K	109%	109%	115%	106%	100%	97%
	14K	109%	109%	114%	106%	100%	98%
	18K	109%	109%	110%	105%	100%	98%
	1.5K	111%	111%	126%	109%	100%	96%
	3K	102%	103%	122%	100%	100%	94%
	6K	102%	103%	127%	101%	100%	95%
EDP	10K	105%	106%	129%	100%	100%	96%
	12K	107%	107%	117%	101%	100%	94%
	14K	104%	106%	115%	101%	100%	96%
	18K	107%	107%	116%	101%	100%	96%

Table 3. Execution cycles, energy, and energy-delay product for EMBEDDED-LE and EMBEDDED-LR contention management policies expressed as percentage of EMBEDDED-LR (abort-based). Experiments done with 6 cores, 60:40 ratio get():set().

References

[1] F. Angiolini, J. Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous MPSoC design space exploration. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 1145–1150, 3001 Leuven, Belgium, Belgium. European Design and Automation Association. ISBN 3-9810801-0-6.

[2] G. Capodanno, R. I. Bahar, D. Papagiannopoulou, T. Moreshet, and M. Herlihy. Embedded-Spec: A light-weight and transparent hardware implementation of lock elision for embedded multicore systems. Submitted to TRANSACT'13.

[3] H. Cho, B. Ravindran, and E. D. Jensen. Lock-free synchronization for dynamic embedded real-time systems. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 438–443, 3001 Leuven, Belgium, Belgium. European Design and Automation Association. ISBN 3-9810801-0-6.

[4] C. Click. Azul's Experience with Hardware Transactional Memory. Retrieved from `http://sss.cs.purdue.edu/projects/tm/tmw2010/talks/Click-2010_TMW.pdf`, 12 December 2012.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGARCH Comput. Archit. News*, (5):336–346. ISSN 0163-5964. doi: 10.1145/1168919.1168900.

[6] EEMBC. Eembc, the embedded microprocessor benchmark consortium. `http://www.eembc.org`.

[7] Eliot and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, (2):186–201, Dec. . ISSN 0167-6423. doi: 10.1016/j.scico.2006.05.010.

[8] C. Ferri, A. Marongiu, B. Lipton, R. I. Bahar, T. Moreshet, L. Benini, and M. Herlihy. SoC-TM: integrated HW/SW support for transactional memory programming on embedded MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '11, pages 39–48, New York, NY, USA, . ACM. ISBN 978-1-4503-0715-4. doi: 10.1145/2039370.2039380.

[9] 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*, (1):47–54, Mar. . ISSN 0163-5964. doi: 10.1145/1241601.1241611.

[10] C. Ferri, S. Wood, T. Moreshet, I. Bahar, and M. Herlihy. Energy and Throughput Efficient Transactional Memory for Embedded Multicore Systems. In Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *High Performance Embedded Architectures and Compilers*, Lecture Notes in Computer Science, pages 50–65. Springer Berlin Heidelberg, . doi: 10.1007/978-3-642-11515-8_6.

[11] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, (124), Aug. . ISSN 1075-3583.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News*, (2), Mar. . ISSN 0163-5964. doi: 10.1145/1028176.1006711.

[14] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http:/http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

[15] J. Lee and K. H. Park. Delayed Locking Technique for Improving Real-Time Performance of Embedded Linux by Prediction of Timer Interrupt. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, RTAS '05, pages 487–496, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2302-1. doi: 10.1109/RTAS.2005.16. URL `http://dx.doi.org/10.1109/RTAS.2005.16`.

[16] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing On-Chip Communication in a MPSoC Environment. In *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, DATE '04, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-2085-5.

[17] memcached. memcached - a distributed memory object caching system. `http://www.memcached.org`.

[18] C. A. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing.

[19] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, pages 606–611, 3001 Leuven, Belgium, Belgium. European Design and Automation Association. ISBN 3-9810801-0-6.

[20] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA. ACM. ISBN 1595934510. doi: 10.1145/1168857.1168902.

[21] M. Pohlack and S. Diestelhorst. From lightweight hardware transactional memory to lightweight lock elision. TRANSACT 2011, 2011.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, . IEEE Computer Society. ISBN 0-7695-1369-7.

[23] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. *SIGOPS Oper. Syst. Rev.*, (5):5–17, Oct. . ISSN 0163-5980. doi: 10.1145/635508.605399.

[24] A. Tumeo, C. Pilato, G. Palermo, F. Ferrandi, and D. Sciuto. HW/SW methodologies for synchronization in FPGA multiprocessors. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '09, pages 265–268, New York, NY, USA. ACM. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508174.

[25] C. Yang and A. Orailoglu. Light-weight synchronization for interprocessor communication acceleration on embedded MPSoCs. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '07, pages 150–154, New York, NY, USA. ACM. ISBN 978-1-59593-826-8. doi: 10.1145/1289881.1289909.

[26] C. Yu and P. Petrov. Latency and bandwidth efficient communication through system customization for embedded multiprocessors. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 766–771, New York, NY, USA. ACM. ISBN 978-1-60558-115-6. doi: 10.1145/1391469.1391665.