Energy-efficient and high-performance lock speculation hardware for embedded multicore systems

Papagiannopoulou, Dimitra

ASSOC COMPUTING MACHINERY

https://doi.org/10.1145/2700097
http://hdl.handle.net/2144/22759

Boston University
Energy-Efficient and High-Performance Lock Speculation Hardware for Embedded Multicore Systems

DIMITRA PAPAGIANNOPOULOU, GIUSEPPE CAPODANNO, TALI MORESHET, MAURICE HERLIHY, R. IRIS BAHAR, Brown University, Boston University

Embedded systems are becoming increasingly common in everyday life and like their general-purpose counterparts, they have shifted towards shared memory multicore architectures. However, they are much more resource-constrained, and as they often run on batteries, energy efficiency becomes critically important. In such systems, achieving high concurrency is a key demand for delivering satisfactory performance at low energy cost. In order to achieve this high concurrency, consistency across the shared memory hierarchy must be accomplished in a cost-effective manner in terms of performance, energy, and implementation complexity. In this paper, we propose EMBEDDED-SPEC, a hardware solution for supporting transparent lock speculation, without the requirement for special supporting instructions. Using this approach, we evaluate the energy consumption and performance of a suite of benchmarks, exploring a range of contention management and retry policies. We conclude that for resource-constrained platforms, lock speculation can provide real benefits in terms of improved concurrency and energy efficiency, as long as the underlying hardware support is carefully configured.

Categories and Subject Descriptors: B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids; B.6.1 [Design Styles]: Memory control and access

General Terms: Design, Performance

Additional Key Words and Phrases: transactional memory, energy-efficiency, low-power, lock elision, lock removal

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The transition of embedded systems towards multicore architectures promises an improvement in power-performance scalability. However, this promise can be realized only if applications are capable of a high enough level of concurrency at low energy cost. Most embedded multi-processor-system-on-chip (MPSoC) designs rely on shared memory for inter-core communication, since it is easier to use than alternatives such as message-passing.

Locks are typically used to guarantee memory consistency in shared memory programs. Locks, however, can limit concurrency and therefore slow performance. They can also be costly in terms of energy. Locks must be deployed conservatively whenever conflicting memory accesses are possible, even when they are unlikely. By contrast, speculative approaches, which detect conflicts dynamically, promise both to improve performance and to save energy. Transactional Memory, Speculative Lock Elision (SLE), and Transactional Lock Removal (TLR) are hard-

This work is supported in part by NSF under Grants CCF-0903384, CCF-0903295, CNS-1319495, and CNS-1319095 as well the Semiconductor Research Corporation under grant number 1983.001.
Authors’ addresses: D. Papagiannopoulou and R. I. Bahar, School of Engineering, Brown University, Providence, RI 02912
M. Herlihy, Department of Computer Science, Brown University, Providence, RI 02912
T. Moreshet, College of Engineering, Boston University, Boston, MA 02215
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1539-9087/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

ware speculation techniques that allow critical sections to execute in parallel, without explicitly acquiring locks [Herlihy and Moss 1993], [Harris et al. 2010], [Rajwar and Goodman 2001], [Rajwar and Goodman 2002]. If a data conflict does take place, it is detected, and one or more of the conflicting threads is rolled back and restarted, either speculatively or by acquiring a lock.

This paper describes EMBEDDED-SPEC, an energy-efficient embedded architecture that supports lock elision, a synchronization mechanism that combines attractive properties of both locking and speculative synchronization. In lock elision, speculation is transparent: when conventional, lock-based software is executed, the underlying hardware attempts to execute certain critical sections as speculative transactions. When all goes well, threads that would have been serialized by a lock can execute that lock’s critical section in parallel, provided there are no data conflicts at run-time. Lock elision is appealing because it promises to increase concurrency without the need to retrofit code.

This paper makes the following contributions:

— Unlike most prior works on speculative synchronization, we focus on energy efficiency as well as throughput since both are key constraints for embedded systems. Specifically, we evaluate the energy-delay product (EDP), a figure of merit that captures the trade-off between these two properties.

— There are many possible hardware designs that could support lock elision. However, because we are considering embedded platforms, which are highly resource constrained, any proposed hardware mechanisms must be simple. We propose the addition of simple hardware structures that avoid changes to the underlying cache coherency protocol but leave us the flexibility to vary how synchronization conflicts are detected, how they are resolved (contention management) and which policy to use for switching between speculative and non-speculative executions.

— We provide a fully transparent solution for speculative execution of locks. This means programmers can take full advantage of the underlying speculative hardware support even when running code written using traditional locks. We evaluate our proposed architecture through a range of benchmarks written with standard locks.

We find that EMBEDDED-SPEC can improve EDP for most of the benchmarks under many of the configurations we consider. The benefits of speculation are sensitive to the critical section size, the degree of lock contention, the retry policy, and the underlying hardware transactional memory’s contention management policy. We conclude that for platforms where energy efficiency matters, EMBEDDED-SPEC can provide real benefits, but that the underlying hardware architecture must be configured with care.

The rest of this paper is organized as follows. We discuss related work in Section 2. Next, Section 3 provides a background discussion on the main speculative memory design components and then describes the EMBEDDED-SPEC architecture, along with its two principal configurations: EMBEDDED-LE (for lock elision), which provides flexible policies for switching between speculative and non-speculative executions, and EMBEDDED-LR (for lock removal), which relies on the progress guarantees provided by the underlying hardware transactional memory to resolve conflicts. Section 4 evaluates these configurations, and their variants, by measuring the EDP of a range of benchmarks. Section 5 summarizes our conclusions and offers directions for future work.

2. RELATED WORK

The authors of [Ferri et al. 2010a] proposed EMBEDDED-TM, an energy-efficient hardware transactional memory design suitable for energy-efficient embedded systems. That design consists of a single L1 cache structure with limited associativity for both storing transactional and non-transactional data. A small, fully-associative victim cache handles overflowed and evicted transactional blocks, and is powered down when not in use. A lazy conflict resolution, though complex to implement, improved the performance of high-conflict workloads, while an eager scheme was a better fit for low-conflict workloads.

In a later paper, [Ferri et al. 2011] proposed an integrated hardware-software transactional memory design for embedded systems. This scheme includes a hardware transactional memory (HTM)
architecture with a dedicated hardware module, the Bloom Module, to handle conflict management, that is programmed through low-level primitives. The Bloom Module manages a centralized collection of Bloom filters. It is in charge of snooping transactional data traffic on the bus, and detecting conflicts that arise during transactions. The Bloom Module is a departure from much prior work by decoupling the transactional memory system from the cache coherence hardware. Using a single hardware device to snoop the shared bus reduces design complexity and enhances portability, since this design does not change the CPU hardware.

While the design introduced in [Ferri et al. 2011] is energy-efficient and simple, it is not transparent to the programmer, who must program using special transactional instructions that enable speculation. In this paper, we explore methods of applying speculation on legacy code without the requirement of special supporting instructions in software. We utilize the Bloom Module hardware, that was introduced in [Ferri et al. 2011], for support of data conflict detection and resolution without altering the cache coherence protocol. Based on this hardware, we introduce a solution that is completely transparent to the programmer.

Rajwar and Goodman in [Rajwar and Goodman 2001] originated the modern study of transparent speculation. They proposed speculative lock elision (SLE), a hardware speculation technique that allows parallel execution of critical sections. The speculative hardware detects when a thread enters and leaves a critical section, elides the delimiting synchronization operations, and executes the intermediate code speculatively, buffering updates. If the critical section completes without a data conflict with a concurrent thread, the updates are committed, and otherwise they are discarded, and the critical section is retried non-speculatively (by actually acquiring and releasing the lock). SLE does not require instruction set changes, coherence protocol extensions, or programmer support. As a result, legacy (even binary) code can run speculatively without modification. The main drawback is the cost of re-executing failed speculative executions.

Rajwar and Goodman [Rajwar and Goodman 2002] later proposed another transparent speculative synchronization mechanism, called transactional lock removal (TLR). Here, conflicts are resolved using timestamps. When a conflict occurs, the conflicting core with the oldest running transaction wins and proceeds with its transaction, while the others are rolled back and suspended until the winning core commits. At that point, the suspended cores resume and re-execute the critical section speculatively. This way there is no need to transition from speculative to non-speculative executions, improving performance while maintaining transparency.

Sun’s Rock [Dice et al. 2009] is an example of a multicore processor implementation that supports some form of best-effort Hardware Transactional Memory. Recently, Intel announced Haswell [Intel Corporation 2012], a new processor microarchitecture with direct hardware support for speculative transactions. Using special constructs in software, programmers can specify regions of the code for either transactional memory or speculative lock elision. Haswell is the first x86 processor that features hardware transactional memory by including Transactional Synchronization Extensions (TSX). Intel’s TSX specification describes how Transactional Memory is exposed to the programmers, but the details on the actual TM implementation were not made public. The TSX specification provides two interfaces to the programmers: The first one is the Restricted Transactional Memory (RTM), which is similar to standard TM proposals. The second one, is Hardware Lock Elision (HLE), which functionality is very close to the initial SLE proposal by [Rajwar and Goodman 2001]. Both of them utilize new instructions to take advantage of the existing TM hardware.

Although the exact details of the lock elision implementation in Haswell are not released, a best effort speculation on its implementation have been discussed [Kanter 2012], [Kleen 2014]. HLE uses two new instruction hint prefixes (XACQUIRE and XRELEASE) to denote the region in the code where lock elision can be applied. When a lock acquisition is encountered in the code, the XACQUIRE prefix is inserted to indicate the start of the lock elision region and the lock instruction is added to the read-set of a transaction, but the lock is not acquired (i.e. the thread does not write new data in the lock address). This means that other threads can also enter the lock elision region simultaneously and transactionally access shared data. Writing to the lock address during execution
of the HLE region will cause an abort. Reads and Writes to shared memory that happen within the lock elision region are added to the Read and Write sets of the corresponding transaction. When the XRLEASE prefix is encountered, it means that the end of the lock elision region has been reached, and the transaction attempts to commit. In a conflict event, the core restores the internal registers state that was saved prior to XACQUIRE and ignores any writes to shared memory that happened within the HLE region. The thread will retry the HLE region again, but this time by normally acquiring the lock. This means that once aborted, no speculation retries are allowed right after. Moreover, there is a limit on the number of simultaneous elisions. If this limit is surpassed, other regions will be executed through standard locking.

Even though Lock Elision was proposed years ago, its main idea enters the mainstream via Intel’s Haswell. The important benefit of Hardware Lock Elision is that it can be used in existing lock-based programs and it takes little programming effort to implement, making the solution suitable for legacy code. Noting the adaptation of Lock Elision in future processor designs and knowing its potential, we propose in this paper efficient alternative algorithms that extend the capabilities of these existing techniques by introducing an extra degree of flexibility and with energy efficiency as an additional primary criteria.

The **EMBEDDED-***SPEC architecture proposed here goes beyond SLE and TLR and the Haswell HLE design in several ways. First, in Haswell and the original SLE proposal of [Rajwar and Goodman 2001], a failed speculation immediately restarts non-speculatively. There are no alternative failover mechanisms or policies. In contrast, our proposal offers flexible **contention management** (conflict resolution) alternatives, including alternatives to TLR’s timestamps. Moreover, the SLE and TLR proposals, like most work in this area, were concerned with improving throughput, not energy efficiency. Since we are concerned with embedded platforms, we take energy-delay product (EDP) to be the principal figure of merit.

Pohlack and Diestelhorst [Pohlack and Diestelhorst 2011] evaluated the results of applying lock elision to the Memcached caching system. Their lock elision implementation was based on AMD’s advanced synchronization facility [Christie et al. 2010] (ASF), a speculative synchronization architecture similar to transactional memory. Our work differs from [Pohlack and Diestelhorst 2011] in that we are concerned with embedded platforms and energy-delay product, instead of general-purpose platforms and throughput.

3. IMPLEMENTATION

3.1. The Speculative Memory

Before getting into the details of our implementation, it is important to describe the basic parameters of designing a speculative memory framework. Here, we present a background discussion on the main components of speculative memory design and how existing transactional memory works used them. Next, we will discuss our design choices and present the proposed algorithms.

Regardless of the implementation that is followed (hardware, software or hybrid), when designing a Speculative Memory scheme, we have to decide on the following important aspects:

- **Conflict Detection.** When and how should a conflict be detected?
- **Conflict Resolution.** When and how should a conflict be resolved?
- **Data Versioning.** Where and how should original values as well as speculative changes to shared data be stored?

Regarding **Conflict Detection** and **Conflict Resolution**, there are two main policies: *eager* and *lazy* conflict detection or resolution. With *eager conflict detection*, conflicts are detected when they occur (i.e., at the time of the data access). The potential problem of this approach is that after a conflict the restarted transactions may abort committing transactions. *Consequent conflicts* can hurt progress. Examples of existing works that use eager conflict detection are [Moore et al. 2006], [Yen et al. 2007] and [Bobba et al. 2008]. On the other hand, in a *lazy conflict detection* scheme, conflict detection is performed at commit time. Potential existing conflicts

with other transactions are detected only when a transaction attempts to commit. This scheme does not have the progress guarantee problem of the eager detection scheme, but since transactions are fully executed until commit time and only then conflicts are detected, this scheme has the drawback of wasted cycles compared to eager conflict detection, since some transactions will continue executing after the conflict actually occurs, only to be aborted when they attempt to commit. This extra useless work execution wastes time and power resources.

Similarly to conflict detection, conflict resolution can happen eagerly or lazily. For eager conflict resolution, the decision of which transaction to abort is made immediately when the conflict is detected. In lazy conflict resolution the decision is postponed until commit time. It is obvious that lazy conflict detection cannot co-exist with eager conflict resolution. Lazy conflict resolution has a throughput advantage [Blundell et al. 2006] but the drawback of additional complexity. Hence it can improve the performance of high-conflict workloads. Previous works like [Shriraman et al. 2010] and [Tomić et al. 2009] used eager conflict detection through the cache coherence protocol and lazy resolution via special hardware and software structures. Overall, eager detection/recovery is easier to implement in hardware through standard cache coherence protocols, but tends to favor short transactions over longer ones.

Another important aspect of conflict resolution is the abort policy in deciding which transactions should be aborted upon a conflict. The requestor-abort policy aborts the transaction that requested the data access that caused the conflict. The rationale behind this is that, since all transactions have made some progress before the requestor caused the conflict, the requestor should be the one to abort so that the other transactions can continue to make progress. Another option is to let the requestor proceed and abort all other conflicting cores (i.e., the requestor-wins policy). This choice is more natural to the way cache coherency works. We could also abort all the transactions that conflicted and let them retry speculation again. Regardless of the chosen abort policy, it is critical that, after a conflict we do not allow all transactions to retry at the same time since this would inevitably result in consecutive aborts. Instead, they should delay retry (or “backoff”) for different randomly chosen times. Many works have used exponential-backoff strategies that increase the backoff time exponentially based on the number of consecutive aborts experienced by each transaction.

Speculative memory requires a means for storing modifications to speculative data while simultaneously keeping copies of the original data, to be able to restore original values in case of conflict. Data Versioning mechanisms determine how this is done. Eager data versioning stores and modifies the speculative data in-place and keeps original data values elsewhere. In this case, the speculative memory system must guarantee a rollback mechanism, usually implemented by means of log structures, to restore the original contents of the memory. This technique has been used in [Ananian et al. 2005], [Moore et al. 2006] and variants like [Yen et al. 2007] and [Bobba et al. 2008]. Keeping speculative data in-place makes commits faster. Since data are updated in-place, no data broadcast is required upon commit. However, it has two drawbacks. First, upon data writes, an extra overhead has to be paid for the original data to be saved into the log. Second, recovery time during aborts is increased, since a complex roll-back mechanism has to be followed in order to read the logged values and restore them, while other transactions are stalled. Hence, an eager versioning scheme should be avoided when high contention is experienced.

In contrast, lazy data versioning leaves old copies of transactional data in-place and creates a copy for speculative modifications in other memory locations or transactional buffers. Examples of designs that have used this policy are [Hammond et al. 2004], [Ananian et al. 2005], [Rajwar et al. 2005], and [Herlihy and Moss 1993]. These designs mostly use their caches to store the speculative data and in some cases extra buffers or software structures to handle overflows. Keeping the original data in their initial location makes the abort scenario very fast, but has the disadvantage of increasing the transaction execution time since since extra time is necessary at commit to write the speculative data back to memory. Some of these lazy versioning schemes ([Herlihy and Moss 1993], [Rajwar et al. 2005], [Ananian et al. 2005], [Ferri et al. 2010a]) can also efficiently handle commits, by using the cache coherence protocol to keep data consistent at the end of the transaction.
The original HTM design that was introduced in [Herlihy and Moss 1993] and later HTM works such as [Ferri et al. 2010a], utilized the cache coherence protocol to assist managing consistency detection. The big drawback of these works is that they generally require extensive changes in the cache coherence protocol to guarantee conflict detection and resolution. Works such as [Ferri et al. 2011] and [Yen et al. 2007] proposed solutions that decouple the HTM design from the cache coherence protocol. In particular, the authors of [Ferri et al. 2011] proposed an HTM design that requires only a few modifications to the cache system and a dedicated hardware module, the Bloom Module, for conflict detection and resolution. The use of an external separate module for conflict detection and resolution alleviates the need for making extensive changes to the cache coherence protocol, thus simplifying cache coherence logic and reducing the number of tag bits required in the caches. Moreover, multiple conflict recovery schemes can be implemented more easily, since the conflict management decisions are no longer made by each core individually but with a single separate module. Using the Bloom Module, we can enable dynamic selection of conflict resolution policies during execution, based on the characteristics of our applications and the experienced abort rate.

Fig. 1 shows the transactional management architecture with the use of the Bloom module. The dark blocks indicate the three additional hardware components that are necessary for transactional management:

1. A new state bit (called Tx bit) for each line of the data cache, which defines whether the data contained in the line is transactional or not.
2. New logic in the cache controller that handles the new transactional accesses and
3. The external Bloom Module.

By borrowing this design and using it for speculative execution both in EMBEDDED-LE and EMBEDDED-LR we keep our hardware design simple.

As implemented in [Ferri et al. 2011] transactional events are triggered through regular read/write operations on memory mapped registers. For example, starting a transaction is done by writing to a special register in the Bloom Module. When the cache controller detects that write, it sets an internal bit to enable the transactional logic. The extra transactional logic of the cache controller has to carry some extra operations while a transaction is executed, as shown in Figure 2. Note that the controller has to handle two special cases. The first one occurs when a line that is accessed transactionally is already in the cache before starting the transaction. In this case, the data would not be retrieved from the L2 memory, hence the Bloom Module would not have the chance to snoop this access on the bus to include it in the transaction’s read and write sets. The cache controller carries the extra responsibility of issuing a bus access to notify the Bloom Module that the cache line is being accessed transactionally. The second special case occurs when a line that is accessed transactionally is being replaced inside the cache. In this case, the cache controller will perform a transaction overflow so that the transaction is able to complete as an overflowing transaction, while
In this section, we describe our target architecture as well as our specific design choices for our implementation of EMBEDDED-SPEC. In particular, we chose a lazy data versioning scheme that uses as a baseline the design proposed in [Ferri et al. 2011]. In addition, we chose to adopt eager conflict detection and resolution as well as a requestor-wins abort policy since they are more natural to the cache coherence protocol.

EMBEDDED-SPEC is based on the architecture proposed by [Ferri et al. 2011], illustrated in Figure 3. We use MPARM [Angiolini et al.], a cycle-accurate, multi-processor simulator developed for embedded system design space exploration, to model and simulate our architecture. It features a configurable number (up to 8) of RISC-like cores, interconnected through a shared bus (AMBA). Each core has private L1 instruction and data caches, kept coherent through a MESI coherence protocol by per-core snoop devices. The shared memory is a two-level, partitioned global address protocol by per-core snoop devices. The shared memory is a two-level, partitioned global address space.
space (PGAS) hierarchy. Specifically, MPARM simulates an architecture that encompasses distinct physical memory banks, globally visible throughout the system. Each core has a small L1 local scratchpad (SPM), accessible without traversing the system interconnect. Remote SPMs can also be accessed directly through the bus, but at the cost of higher latency. The overall L1 shared memory is the union of the SPMs, and it is globally non-coherent: its addresses are not cacheable, and it is explicitly managed by software. L2 shared memory physically consists of a single device, logically partitioned into a large shared segment, plus small “private” segments for each core. Addresses belonging to the logically shared chunk are cacheable and globally coherent. The private segments are also cacheable, but their addresses are not involved in coherence traffic.

Non-speculative synchronization is supported by a fixed set of architectural hardware locks drawn from a pre-allocated section of memory, the semaphore memory, and accessible by standard synchronization calls such as Test(), TestAndSet(), and Release().

3.3. The Bloom Module Hardware

The Bloom Module [Ferri et al. 2011] is in charge of conflict detection and resolution. It monitors all transactional accesses, records them as per-core signatures, and notifies the CPUs when data conflicts occur. As explained in more detail below, the Bloom Module used in this paper departs from prior designs by also snooping on the semaphore memory.

To support EMBEDDED-SPEC, we extended the Bloom module’s control logic as well as its individual Bloom filters to make it aware of the architecture-supported hardware locks. Fig. 4 shows an overview of the Bloom Module and the internal details of a core Bloom Filter Unit (BFU). The Bloom module has the following functional blocks:

— **Snooping Shared Memory Address**: Snoops the shared memory address space to keep track of the addresses accessed during speculative execution.
— **Bloom Filters**: Per-core signatures corresponding to the read and write addresses accessed during the speculative execution.
— **Control Logic**: Implements the features needed to manage communication with the cores (i.e., the abort and hold signals). It also manages the abort policies and handles cache overflow.
— **Snooping Semaphore Memory Address:** Snoops traffic to and from the hardware locks. It detects Test(), TestAndSet() and Release() calls and their responses.
— **SLE Registers:** Per-core registers to keep track of the core status (i.e., which core is in speculative mode on which hardware lock and which core has ownership of a specific lock). These registers are kept updated by the Snooping Semaphore Memory Address block.
— **Hold Queue List Registers:** Per-core registers to keep the list of aborted cores that need to be released at commit time. These registers are used only with EMBEDDED-LR.

Communication between the cores and the Bloom Module is handled via interrupts and read/write memory operations (no extra wires are required). A small memory space (approximately 256 Bytes) is reserved for programming registers, used to program specific functionalities of the Bloom Module at run time. For example, the commit priority is set using read and writes to specific registers in this set. As seen in Fig. 4, each core has a Bloom Filter Unit (BFU) consisting of K read-write pairs of simple Bloom filters. Instead of setting multiple bits in one large filter, our design sets a single bit in K small bloom filters. This parallel Bloom filter design limits the required hardware. Empirical experimentation in [Ferri et al. 2011] showed that K=4 provides the best power/performance tradeoff. Moreover, the hash functions were designed with delay and power as the main criteria and it was thus decided to implement the hash function on a single level of two input XORing of lower order address bits. \(^1\)

### 3.4. The EMBEDDED-SPEC Algorithms

We examined two variations of the EMBEDDED-SPEC architecture:

1. **EMBEDDED-LE (Embedded Transparent Lock Elision):** The critical section is executed speculatively by eliding the lock. The Bloom module monitors memory accesses, and if there is a data conflict, it directs the conflicting cores to roll back their speculative executions and contend for the lock. One will succeed, and the rest will spin until the winner releases the lock. When the lock is released, the waiting cores retry their speculative executions. If the number of retries for a specific transaction (due to repeated conflicts) exceeds a threshold, the cores revert to non-speculative execution for that instance of the transaction. When the end of the critical section is successfully reached, the number of retries is reset to zero.

2. **EMBEDDED-LR (Embedded Transparent Lock Removal):** As with EMBEDDED-LE, the critical section is executed speculatively by eliding the lock, but in case of a data conflict, the Bloom module directs all conflicting cores but one (the winning core), to roll back and suspend execution until the active core completes the critical section. When the winner completes, the suspended cores resume speculative execution, so a lock never needs to be explicitly acquired.

EMBEDDED-LE supports two contention management policies. The requester-abort policy aborts only the core requesting the conflicting address, and the abort-all policy aborts all cores executing the same critical section. The second policy is motivated by the observation that once a core abandons speculation and tries to acquire the lock, it is highly likely it will force the other cores in the same critical section to abort eventually.

We also examined a variation of EMBEDDED-LE, in which cores suspend execution in a low-power idle mode instead of spinning when waiting for a lock. This approach (called EMBEDDED-LE-SLEEP) saves power but increases latency (by 2 ms). Finally, we examined the effects of allowing aborted cores to attempt to elide the lock more than once before resorting to lock mode by setting a parameter \(\text{max number of retries}\).

In EMBEDDED-LR, cores never switch from speculative to non-speculative execution. Unlike “best-effort” HTMs, EMBEDDED-LR guarantees that every transaction eventually commits so EMBEDDED-LR is not subject to starvation. EMBEDDED-LR supports two abort policies: \(\text{times-}\)

---

\(^1\) This was based on a previous finding that the lower order bits of an address are characterized by more randomness than the higher order bits.
tamp and priority-abort. For the timestamp policy, the core with the earliest timestamp is allowed to proceed, whereas in the priority-abort policy, each core has a priority that is increased when it is rolled back and in case of conflict, the higher-priority transaction proceeds. Table 3.4 summarizes all the possible configurations for EMBEDDED-LE and EMBEDDED-LR. The two algorithms are discussed in more detail next.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Abort Policy</th>
<th># retries</th>
<th>Sleep</th>
</tr>
</thead>
</table>
| EMBEDDED-LE   | 1) requestor-abort  
               2) abort-all      | 0,1,2,...,∞  | Yes/No |
| EMBEDDED-LR   | 1) timestamp  
               2) priority-abort | N/A        | Yes/No |

Table 3.4: EMBEDDED-SPEC — All Configurations.

3.4.1. EMBEDDED-LE. Figure 5 shows the flowchart of the EMBEDDED-LE algorithm, which is implemented in middleware using API function calls and hardware lock instructions (i.e. Test(), TestAndSet()).

This algorithm (Figure 5) is called when a core tries to enter a critical section protected by a lock. For example, when core X tries to enter a critical section, it checks whether the maximum number of retries has been exceeded. If so, it calls TestAndSet() to try to acquire the lock. If not, it calls Test() to determine whether the lock is free. If the lock is not free, the core spins until the lock is free. When the lock is observed to be free, the core elides it, and executes the critical section speculatively.

The Bloom module detects and resolve data conflicts during speculative executions. The affected cores will have their states restored, and the Tx_abort register will be updated to indicate the abort occurred. The core calls the Check_abort() function to determine whether it has been aborted. If not, the core proceeds speculatively. If so, the core terminates the speculative execution, and calls TestAndSet() once to try to acquire the lock. If it succeeds, the core proceeds non-speculatively. Otherwise, the core returns to the retries count checkpoint to determine whether it should continue to speculate, or fall back to locking. Either way, the core spins until the lock is freed.

If eventually core X reaches the end of its critical section, it will check its own execution mode by calling Check_In_Transaction, to determine whether it has been running in locking mode or speculative mode in order to either release the lock or end the speculative execution.

3.4.2. EMBEDDED-LR. EMBEDDED-LR requires extensions to the Bloom module, and small changes to the middleware, replacing each lock acquisition with a start_transaction instruction. Once a core starts speculation, it will never try to acquire the lock, even in case of a conflict. When a conflict is detected, the losing cores are suspended.

We note that the EMBEDDED-LR algorithm does not require much support in the middleware level, since the Bloom Module is already present in the hardware level. The only required feature at the middleware level is starting a new transaction instead of acquiring the lock. Even in the event of a mis-speculation, the lock will not be acquired. The idea behind this is to allow at least one core to complete the critical section. In case of a conflict, the core or cores that have been selected to stop (i.e., the “losing” cores), will be aborted and put in a hold state by the Bloom module. When the ‘winning’ core completes execution of its critical section, the core or cores kept in a hold state, will be released and will be allowed to retry the critical section speculatively. To track suspended cores, the Bloom module is extended with a per-core hold_queue_list. When a core is aborted, its CoreID is added to the winning core’s hold_queue_list register. We also append to the winning core’s hold_queue_list register the hold_queue_list lists of the aborted cores. When the winning core commits, every CoreID in its hold_queue_list register is released and the list is cleared.
4. EXPERIMENTAL RESULTS
In this section we evaluate the proposed EMBEDDED-SPEC design. We tested our architecture with several configurations. The first part of the evaluation is devoted to finding the optimal
parameters in terms of energy consumption and execution time. Our target metric is therefore energy delay product (EDP), which is a standard commonly used evaluation metric in computer architecture [Horowitz et al. 1994]. As it was done in previous work using the MPARM simulator [Ferri et al. 2010b], the performance and power models are based mostly on data obtained from a 0.13 \textmu m technology provided by STMicroelectronics for their Nomadik Platform [STMicroelectronics 2008], and the energy model for the fully associative caches is based on [Efthymiou and Garside 2002]. The second part of the evaluation is focused on evaluating the advantages of the optimal configuration over the baseline lock approach. The hardware parameters are reported in Table 4.

<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>8KB 1-way Icache, 16KB 4-way Dcache</td>
</tr>
<tr>
<td>Cores</td>
<td>{1, 2, 4, 8}</td>
</tr>
<tr>
<td>Policies</td>
<td>Locking, EMBEDDED-LE, EMBEDDED-LR</td>
</tr>
<tr>
<td>Signature</td>
<td>2KBits 4-way, Read and Write Bloom filters</td>
</tr>
</tbody>
</table>

Table 4: Hardware configurations.

4.1. Benchmarks

To test our design, we chose several benchmarks that were adapted to our simulation platform, which does not include operating system support. The benchmarks belong to the following suites:

— The STAMP benchmark suite [Minh et al. 2008]. The selected workloads represent the following synchronization patterns and critical section sizes: 1) large non-conflicting critical sections (vacation); 2) barrier-based synchronization with small critical sections (kmeans); 3) large critical sections that may conflict (genome); 4) a mix of large and small critical sections (labyrinth).
— The MiBench suite [Guthaus et al. 2001] patricia: A Patricia trie is a data structure used in place of full trees with very sparse leaf nodes. Patricia is characterized by a high percentage of time spent in critical sections, and a high abort rate.
— Datastructures. redblack, skiplist: applications operating on special data structures. 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.

For our experiments, we start with a design space exploration using the set of benchmarks described above. From this design space exploration, we determine the best combination of abort and retry policies for the two EMBEDDED-SPEC algorithms. Next, we compare our best configurations against standard locks.\(^2\)

4.2. EMBEDDED-LE Parameter Exploration

Sleep Mode. As described in Section 3.4, the EMBEDDED-LE implementation can be executed in conjunction with sleep mode, where if a thread is unable to acquire a lock immediately, it is switched to an IDLE state to reduce energy consumption. The energy savings, however, come at the expense of an increased execution time required to switch the cores back from sleep mode to normal execution.

\(^2\)Note that in this work we are not focusing on showing the EDP improvement achieved by speculative techniques as we increase parallelism by scaling the number of cores, since this has already been demonstrated by previous work (e.g., [Ferri et al. 2010a]). Instead, our emphasis is on providing a detailed exploration of a range of contention management techniques and retry policies and comparing the EDP improvement achieved specifically based on those choices. Hence we do not normalize our results to the single thread execution, but rather we compare them to the base synchronization approaches.
operation. Fig. 6 shows how the execution time is affected by including sleep mode execution along with lock elision (noted as EMBEDDED-LE-SLEEP) as the number of cores is varied.\textsuperscript{3}

To explain the differences observed in the results of the aforementioned benchmarks, we need to bear in mind what is special about each one of them. The genome, patricia and vacation benchmarks have large critical sections and they spend a significant portion of time executing critical sections. While patricia experiences high abort rates, vacation generally has non-conflicting transactions. The redblack and skiplist benchmarks are very similar in the sense that they both work on special data structures and have very low abort rates. The labyrinth benchmark includes a mix of large and small critical sections. Finally, kmeans is a benchmark that spends a very small portion of its execution time in critical sections whose size are very small. That is why kmeans often does not show significant changes in behavior when fine-tuning some parameters.

As seen in Fig. 6, all benchmarks except for patricia show an increase in execution time.\textsuperscript{4} This increase is usually negligible and below 5%, but for vacation and redblack it reaches up to 6% and 10% respectively. This increase is expected since switching to/from sleep mode imposes a small time overhead (0.2 $\mu$s, i.e., 40 cycles). Only patricia shows a decrease in execution time of 4%. We believe that this happens because the small latency introduced by switching to sleep mode can shift timing in such a way that by the time sleeping cores wake up and retry speculation, the cores they previously conflicted with have completed their critical sections so they don’t conflict again. For benchmarks such as patricia, which have relatively high abort rates, a timing shift can have a big impact on the resulting abort rate and hence on performance. Indeed, in this experiment the abort rate for patricia decreased from 42% to 37% when using sleep mode.

Moreover, Fig. 7, which reports the energy consumption for the same set of experiments, shows that for benchmarks that spend a considerable amount of time in critical sections, there is a significant reduction in energy consumption due to sleep mode (e.g., 18% for genome and reaches 48% for patricia). Only redblack shows a slight increase (3%), while kmeans, skiplist and labyrinth are not affected at all by sleep mode. Since redblack has a very low abort rate, sleep mode only adds extra energy overhead.

In Fig. 8 we show the energy-delay product for the same set of experiments in order to measure the combined effect of sleep modality on both performance and energy consumption. Even though execution time is increased for some benchmarks when sleep mode is used, the effect is largely compensated by the reduction in energy consumption, resulting in a significant EDP improvement in most cases, reaching 14% for genome, 50% for patricia and 20% for vacation. The overall effect of sleep mode in EDP is insignificant for skiplist and non-existent for kmeans and labyrinth. Only redblack shows a clear decrease in EDP from sleep modality for the same reasons mentioned before.

\textsuperscript{3}The labyrinth benchmark triggers software generated transaction aborts, which we do not currently support. We therefore omit the simulations where these are triggered (4 and 8-core configurations for EMBEDDED-LE).

\textsuperscript{4}Note that for most of the figures shown, the y-axis is not 0-based, in order to make the observed trends more readable.
The conclusion we draw from this set of experiments is that if we care only about performance, we should use EMBEDDED-LE-SLEEP modality instead of EMBEDDED-LE for patricia while we should avoid it for all other benchmarks. If we care only about energy consumption, then EMBEDDED-LE-SLEEP modality is overall a better choice. Similarly, if we care for both performance and energy consumption, then, overall, EMBEDDED-LE-SLEEP is the better way to go.

So far, we showed that for many benchmarks it is better to sleep instead of spin. However, to better understand the design space, in the following sections we will continue our parameter exploration testing both sleeping and spinning versions of each configuration.

**Max Number of Retries.** Note that in all our experiments for EMBEDDED-LE so far, once a thread failed to elide a lock, it would then try to acquire it. We next extend EMBEDDED-LE to allow a thread that had a conflict during lock elision to retry eliding the lock rather than immediately trying to acquire it. Therefore, the next parameter we investigate for EMBEDDED-LE is the max number of retries, which allows us to evaluate how many times it is worthwhile to retry a failed speculation on a high conflict critical section before we switch back to lock mode.

Fig. 9 shows the performance with a varying number of retries allowed before reverting to locks, in EMBEDDED-LE mode. Note that by setting this value to 0, EMBEDDED-LE behaves as in prior experiments, acquiring the lock after a single abort. Most benchmarks benefit in terms of performance from retrying the speculation several times, instead of not retrying at all (i.e., having number

---

5Note that in the summary of our results, we are focusing mostly on describing the trends for the 8-core execution, since that is when we experience the most parallelism. Of course, the trends for every number of cores configuration can be observed in detail in all the figures included.

6[Rajwar and Goodman 2001] effectively implemented their version of SLE with max number of misspeculations set to 1, i.e., maximum number of retries set to 0.
Fig. 9. Performance of EMBEDDED-LE and varying maximum number of retries.

of retries set to 0). In particular, when the maximum number of retries is 0, performance generally tends to degrade as the number of cores increases. A limit of 4 is optimal for patricia and genome, but the rest of the benchmarks do not show significant change in performance based on which of the non-zero values we choose (vacation being the only exception, which shows a clearly worse performance if we restrict the number of retries to 1 instead of allowing more than 1 retry). Because they both experience high contention, patricia and genome do not benefit from many retries. In benchmarks with high abort rate, switching to locking is preferable after a few retries, since speculation is likely to fail again. Indeed, for patricia as the number of cores increases we have to make sure the abort rate does not increase to the point where it is counterproductive for performance. When we restrict the number of retries to 0, the abort rate is reduced to nearly 0, but very little thread parallelism is exploited for 4 or 8 cores. If we allow one retry, the abort rate reaches 42% (for 8 cores), but is still tolerable when it comes to improving performance. The same trend is experienced in genome as well. Restricting the number of retries to 0 gives a nearly zero abort rate, while allowing one retry yields abort rate 17%. For these two benchmarks that have highly contended critical sections it is better to limit the number of retries, in order to prevent the abort rate from increasing to the point where it hurts performance. Note that the exact same phenomenon is experienced in sleep modality.

For energy consumption shown in Fig. 10, again 0 becomes the worst choice as we increase the number of cores, but choosing between 2, 4 or an infinite number of retries does not make much difference for most benchmarks. Retrying 4 times seems again to be slightly better for genome and patricia. For the energy-delay product, Fig. 11 shows that picking any non-zero number of retries will yield similar benefits for most benchmarks, except for patricia and genome, where restricting the maximum number of retries to 4 is clearly better (shows 10% EDP improvement). If we had to choose a single maximum retry value to use for all benchmarks, we conclude that retrying up to 4 times would be overall the best choice.

The results for EMBEDDED-LE-SLEEP appear in Fig. 12 and Fig. 13. In this case the results are similar, with a few notable differences. As in the non-sleep case, for most benchmarks, any non-zero number of retries yields similar results in terms of performance. Especially for genome and patricia, retrying 2 times at most is better for performance than not restricting the number of retries. As in the EMBEDDED-LE case, retrying speculation instead of switching back to locks immediately after an abort is always beneficial for performance. When looking at energy though, things change significantly, as now we are able to save considerable amounts of energy while waiting on the lock in sleep mode instead of directly retrying speculative execution. In contrast, the more we allow retrying speculation, the more we risk wasting energy, as Fig. 13 shows. For all benchmarks (except kmeans, which does not spend enough time executing critical sections to matter), restricting the max number of retries to 0, yields considerable energy savings.
Fig. 10. Energy Consumption of EMBEDDED-LE and varying maximum number of retries.

Fig. 11. Energy Delay Product of EMBEDDED-LE and varying maximum number of retries.

Fig. 12. Performance of EMBEDDED-LE-SLEEP and varying maximum number of retries.
To determine the overall best choice, we have to look at the energy-delay product, as shown in Fig. 14. Benchmarks such as redblack, skiplist, kmeans and labyrinth show better results when choosing any non-zero number of retries, while vacation shows considerable improvement (23%) for an infinite number of allowed retries compared to just 1. Genome shows better EDP when we restrict the number of retries to 2. On the other hand, patricia seems to benefit greatly both in performance and energy when we do not allow any retries at all. This is expected, since benchmarks with high abort rates, such as patricia, benefit from switching to locks following a single misspeculation, while benchmarks with lower conflict levels benefit from retrying the speculation several times.

We conclude that if we want to increase performance and at the same time decrease energy consumption, then for most benchmarks (except for patricia and genome), we should allow retrying speculation for an unlimited number of times until it is successful, instead of switching back to locks. Slight variations to the best non-zero value choice are observed, that lead us to pick non-infinite values especially for genome (4 in EMBEDDED-LE mode and 2 in EMBEDDED-LE-SLEEP mode) and patricia (4 in EMBEDDED-LE mode). The only exception to these observations is patricia in EMBEDDED-LE-SLEEP mode. In this case, we see a significant improvement when we do not allow any retries and we immediately switch back to locks after an unsuccessful speculation attempt. Again, this is due to the relatively high contention rate for this benchmark.

If our primary goal is to improve performance, allowing an infinite number of retries is best for all benchmarks, except for genome and patricia, which show better performance for 4 maximum retries in EMBEDDED-LE mode and 2 in EMBEDDED-LE-SLEEP mode. If our primary goal is to decrease energy consumption, then not restricting the number of retries is again best for most benchmarks, except for patricia and genome, which yield better results if we restrict the number of retries to 4. Finally, if we want to decrease energy consumption, but we are in EMBEDDED-LE-SLEEP mode, then we should not allow any retries for any of the benchmarks.

Abort Policy. We continue our parameter exploration by experimenting with the the abort policy, which is set within the Bloom module abort manager. We compare the requestor-abort policy, which aborts only the requesting core when a conflict occurs, to the all cores on the same lock ID policy (or abort-all policy), which aborts all cores conflicting on the same lock-protected critical section.

Note that for either abort policy, the aborted cores will have to explicitly acquire a lock once they have rolled back and restored their previous states. Since multiple cores attempting to execute critical sections on the same lock ID must be consistent (i.e., cores must all be executing either in speculative (LE) or non-speculative (lock) mode), in the case of the requestor-abort policy, the other cores will have to abort as well if the requestor core manages to acquire the lock before they commit. However, since the process of rollback can take several cycles, in many instances the non-aborted
cores will commit before the lock is acquired, and therefore it would have been wasteful to abort all the cores immediately when the conflict was first detected.

We set the maximum number of allowed retries to different values, in order to see if the abort policy plays a different role in each case. We chose three different values: 0, 1, 2 and an infinite number of maximum allowed retries and for each of these values, we tested the two abort policies mentioned before. In all three experiments, the observed trends on performance, energy and EDP were the same, so here we will present only the results for EDP that combines both metrics.

We found that if we do not allow any retries on a failed speculation, then both abort policies yield the exact same results in performance, energy and EDP for all benchmarks, whether we enable sleep mode or not. This is expected since the abort policy does not really create much of a difference if we immediately switch back to locks on an event of a failed speculation.

Fig. 15 shows EDP results for the two abort policies in EMBEDDED-LE mode when we allow at most one speculation retry. As we can see for all benchmarks, both abort policies show similar results, except for genome and patricia, which show considerable benefits for the requestor-abort policy compared to the abort-all policy, as the number of cores increases (18% and 10% improvement respectively). If we allow 2 retries in EMBEDDED-LE mode, as seen in Fig. 16, patricia is consistently better (21%) for requestor-abort, while genome does not show any difference in this case. Vacation also shows a great benefit (19%) for 8 cores. We note though, that as we increase the maximum allowed number of retries, the benefits of choosing the requestor-abort policy become more prominent. Fig. 17 shows the corresponding results for an infinite number of allowed retries in EMBEDDED-LE mode. In this case, we observe a dramatic drop in the EDP for specific benchmarks, like genome, patricia and vacation (47%, 75% and 63% respectively) as we increase the number of cores. Our conclusion from this set of experiments, is that for most benchmarks the abort policy does not affect the overall EDP, but for genome, patricia and vacation, the requestor-abort policy shows significant benefits that become more prominent as we increase the number of maximum allowed speculation retries. Thus, we conclude that we could safely choose the requestor-abort policy every time we execute in EMBEDDED-LE mode.

Next, we repeat the same set of experiments, but this time with sleep mode enabled. Figures 18, 19 and 20 show the corresponding results. We generally observe similar trends as in the non-sleep modality, with the following differences. When we limit the maximum number of retries to 1, as shown in Figure 18, we see that the requestor-abort policy is slightly worse for vacation as we increase the number of cores, but still it is slightly better for genome and patricia. The differences observed in this case though are too small to draw a conclusion on which technique is better. As we move to higher numbers of allowed speculation retries, as shown in Figures 19 and 20, the benefits of the requestor-abort policy become more visible in specific benchmarks. In particular, the EDP
Energy-Efficient and High-Performance Lock Speculation Hardware for Embedded Multicore Systems

Fig. 15. Energy delay product for EMBEDDED-LE using different abort policies and maximum number of allowed retries set to 1.

Fig. 16. Energy delay product for EMBEDDED-LE using different abort policies and maximum number of allowed retries set to 2.

Fig. 17. Energy delay product for EMBEDDED-LE using different abort policies and maximum number of allowed retries set to infinity.
Fig. 18. Energy delay product for EMBEDDED-LE-SLEEP using different abort policies and maximum number of allowed retries set to 1.

Fig. 19. Energy delay product for EMBEDDED-LE-SLEEP using different abort policies and maximum number of allowed retries set to 2.

reduction in the EMBEDDED-LE-SLEEP experiment, when setting the number of allowed retries to infinity and using requestor-abort policy, is 43%, 80% and 76% for genome, patricia and vacation respectively, while it was 47%, 75% and 63% for the EMBEDDED-LE experiment set.

We conclude that it is never disadvantageous to choose the requestor-abort policy over the abort-all policy. In fact, for some benchmarks like genome, patricia and vacation, the requestor-abort policy is beneficial both in terms of performance and energy consumption, especially when we set a higher number of maximum allowed retries and we choose sleep modality.

4.3. EMBEDDED-LR Parameter Exploration

In this section we evaluate the abort policies of the EMBEDDED-LR implementation. As described in Section 3, this approach is distinct from EMBEDDED-LE because the architecture does not use locks for mutual exclusion.

Abort Policy. The abort policies we evaluate are timestamp, which aborts the core with the latest time stamp (i.e., the last core to start executing this critical section) and priority-abort, which favors the core that has been aborted the largest number of times on this particular critical section. To implement a timestamp configuration without increasing the hardware complexity, at the start of a new transactional execution the Bloom module increments a global counter and stores its value in the related Bloom module core register. In this way, each core that is working in speculative mode will keep information about its starting order. When a conflict is detected, the Bloom module aborts the core with the highest value. To implement a priority-abort configuration, the Bloom module...
Energy-Efficient and High-Performance Lock Speculation Hardware for Embedded Multicore Systems

Fig. 20. Energy delay product for EMBEDDED-LE-SLEEP using different abort policies and maximum number of allowed retries set to 100.

Fig. 21. Energy delay product for EMBEDDED-LR using different abort policies.

increments a per-core register every time the core aborts. The register is cleared on commit. Note that in both cases, the aborted cores are switched into sleep mode for energy-saving reasons.

Fig. 21 shows that the timestamp approach provides similar EDP to priority-abort for genome, kmeans, redblack, skiplist and vacation. However, for patricia and labyrinth we observe that the timestamp approach provides significant EDP improvement compared to priority-abort (up to 18% and 31% respectively).

The results for performance and energy consumption show the exact same trends as EDP, so we do not include these graphs, but summarize the results. For performance, timestamp is clearly better than priority-abort for patricia (10%) and labyrinth (23%). For energy consumption, timestamp shows similar improvements for patricia (<10%) and labyrinth (18%). These benchmarks all have some longer running critical sections and therefore they tend to benefit from letting them run to completion, as the timestamp configuration allows. For all other benchmarks, no significant difference is observed between the two policies. Based on these observations, we conclude that we could safely choose the timestamp approach for execution in EMBEDDED-LR mode.

4.4. Speculative Execution vs. Locks

Having determined the optimal set of parameters for each benchmark, we are now ready to compare our implementation of EMBEDDED-SPEC with the standard lock approaches (lock and lock-sleep). Using the best parameter configurations for each execution mode and each benchmark presented so far, we perform a set of experiments in which we compare the performance, energy consumption and EDP of each applied technique (locking, lock-sleep, EMBEDDED-LE, EMBEDDED-LE-SLEEP and EMBEDDED-LR).
Fig. 22. Execution time of EMBEDDED-SPEC vs. standard locks. Showing results for best configurations for each benchmark.

Fig. 23. Energy Consumption of EMBEDDED-SPEC vs. standard locks. Showing results for best configurations for each benchmark.

Fig. 22 shows the execution time of each technique, normalized to the execution time of standard locks. As can be seen, for the 1-core configuration locks provide better performance than any kind of speculation. This is expected, and is due to the additional hardware and software support necessary to enable the speculation. As we increase the number of cores though, the speculative approaches begin to show an advantage for all but the kmeans benchmark. As mentioned earlier, in kmeans the critical sections are rare and small (i.e., less than 5% of time is spent in critical sections), and the results show that EMBEDDED-SPEC does not provide benefits. At the same time, EMBEDDED-SPEC does not hurt performance when the benchmark does not include large speculative sections.

We also observe that EMBEDDED-LR yields the best performance for an increased number of cores, except for kmeans. EMBEDDED-LR yields performance improvement of at least 47% for patricia and up to 80% for genome, compared to standard locks. The next best configuration to EMBEDDED-LR is EMBEDDED-LE and EMBEDDED-LE-SLEEP, both yielding performance improvement of 10%, 31%, 45%, 50% and 70% for vacation, redblack, skiplist, labyrinth and genome respectively. The only exceptions are kmeans, for the reasons mentioned, and patricia, which shows better performance for the lock-sleep and locking techniques than for EMBEDDED-LE and EMBEDDED-LE-SLEEP. This is expected for patricia, since it suffers from a relatively high abort rate, hence using locking instead of speculation is preferable. Regarding performance, EMBEDDED-LE and EMBEDDED-LE-SLEEP show a very small difference, with EMBEDDED-LE...
being slightly better, apart from patricia and redblack where the difference is more pronounced (23% and 6% respectively). Regarding locking compared to lock-sleep, the difference in performance is again insignificant, apart from patricia, where lock-sleep is clearly better (11%).

Fig. 23 shows the energy consumption for the same set of experiments. Here, lock-sleep is clearly preferable, showing energy benefits starting from 15% for kmeans and reaching up to 73% for genome and labyrinth. When we focus only on energy, locking with sleep mode enabled is clearly better than speculation since it does not encounter aborts. On the other hand, locking without sleep mode enabled becomes the worst choice for energy consumption as we can see in Fig. 23. So, with best choice being the lock-sleep technique, the second best choice in terms of energy consumption is EMBEDDED-LR (except for patricia where EMBEDDED-LE-SLEEP is 31% better than EMBEDDED-LR). EMBEDDED-LE-SLEEP comes very close to EMBEDDED-LR in terms of energy consumption, with EMBEDDED-LE following next for most cases. A common observation is that all sleep techniques yield better energy results, which is generally expected.

Finally, in Fig. 24, we show the combined performance-energy consumption results and make the following observations. First, we see that EMBEDDED-LR and lock-sleep are the two best techniques when we care for both performance and energy consumption, with EMBEDDED-LR being better than lock-sleep for genome, redblack and vacation (up to 19%) and very similar for skiplist, labyrinth and kmeans. Only for patricia lock-sleep is better than EMBEDDED-LR (up to 12%), again because of its high abort rate. The next best configuration for EDP for most benchmarks is EMBEDDED-LE-SLEEP with EMBEDDED-LE being very close. The worst choice for EDP is again locking. There are exceptions: First, kmeans does not show any significant difference for any of the applied techniques, which is expected as explained earlier. Second, patricia is the only benchmark that shows a clear improvement in EDP for EMBEDDED-LE-SLEEP compared to EMBEDDED-LE (73%). Overall, with respect to energy-delay product, EMBEDDED-LR is the best choice, with lock-sleep following next.

We draw the following conclusions: If reducing energy consumption is our primarily goal then we should use sleep-enabling techniques. Moreover, we should not bother using speculation, but choose the lock-sleep technique instead. Speculation is encouraged only in cases where we encounter increased parallelism. On the other hand, if performance is our primary goal then EMBEDDED-LR is clearly the winner. Finally, to improve the energy-delay product, we should generally pick EMBEDDED-LR or lock-sleep and avoid EMBEDDED-LE or traditional locking.

Table 4.4 summarizes the best and second best configuration modes for each of the benchmarks we considered in our experiments. As we discovered, the best configurations may vary depending on whether we only care about performance, energy, or a combination of the two.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Best for Performance</th>
<th>Best for Energy</th>
<th>Best for EDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. SLE-sleep, 2 retries, requestor-abort</td>
<td>2. TLR, timestamp</td>
<td>2. SLE-sleep, 2 retries, requestor-abort</td>
</tr>
<tr>
<td>Kmeans</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. No diff. on type, #retries or abort policy</td>
<td>2. TLR, timestamp</td>
<td>2. lock-sleep</td>
</tr>
<tr>
<td>Patricia</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. SLE, 4 retries, requestor-abort</td>
<td>2. SLE-sleep, 4 retries, requestor-abort</td>
<td>2. lock-sleep</td>
</tr>
<tr>
<td>Redblack</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. SLE, No diff. on #retries or abort policy</td>
<td>2. TLR, timestamp</td>
<td>2. lock-sleep</td>
</tr>
<tr>
<td>Skiplist</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. SLE, No diff. on #retries or abort policy</td>
<td>2. locking</td>
<td>2. TLR, timestamp</td>
</tr>
<tr>
<td>Vacation</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. SLE, No diff. on #retries requestor-abort</td>
<td>2. TLR, timestamp</td>
<td>2. SLE, No diff. on #retries, requestor-abort</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>1. TLR, timestamp</td>
<td>1. lock-sleep</td>
<td>1. TLR, timestamp</td>
</tr>
<tr>
<td></td>
<td>2. No diff. on type, #retries or abort policy</td>
<td>2. TLR, timestamp</td>
<td>2. lock-sleep</td>
</tr>
</tbody>
</table>

Table 4.4: EMBEDDED-SPEC – Top Best two configurations when considering performance only, energy only, or energy-delay product.

5. CONCLUSIONS

We described a lightweight implementation of lock elision on an embedded architecture. Our implementation provides a transparent means of combining attractive properties of both locking and speculative synchronization. Results show energy and performance benefits of our proposed scheme, especially for larger number of cores (e.g., 4–8 cores). In comparing our two schemes (lock elision vs. lock removal), EMBEDDED-LR provides better performance and energy characteristics than EMBEDDED-LE. However, we note that standard locks with sleep mode enabled may still be the best choice if minimizing energy consumption is more critical than improving performance.

In our study, we were careful to preserve the locks and critical sections of the original benchmarks when we ported it to our embedded platform. For future work, we plan to investigate the degree to which simple refactoring of known hot-spots in the code can enhance the benefits of lock elision. In addition, we plan to evaluate the behavior of the two speculative schemes over a wider set of benchmarks, that include more LockID values and broader synchronization patterns.

REFERENCES


Energy-Efficient and High-Performance Lock Speculation Hardware for Embedded Multicore Systems


M. Pohlack and S. Diestelhorst. 2011. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *TRANSCA.*


Received mm yyyy; revised mm yyyy; accepted mm yyyy