Software Transactional Memory for Multicore Embedded Systems

Jennifer Mankin
Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts, USA
jmankin@ece.neu.edu

David Kaeli
Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts, USA
kaeli@ece.neu.edu

John Ardini
Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts, USA
jardini@draper.com

Abstract

Embedded systems, like general-purpose systems, can benefit from parallel execution on a symmetric multicore platform. Unfortunately, concurrency issues present in general-purpose programming also apply to embedded systems, protection from which is currently only offered with performance-limiting *coarse-grained locking* or error-prone and difficult-to-implement *fine-grained locking*. Transactional memory offers relief from these mechanisms, but has primarily been investigated on general-purpose systems. In this paper, we present *Embedded Software Transactional Memory* (ESTM) as a novel solution to the concurrency problem in parallel embedded applications. We investigate common software transactional memory design decisions and discuss the best decisions for an embedded platform. We offer a full implementation of an embedded STM and test it against both coarse-grained and fine-grained locking mechanisms. We find that we can meet or beat the performance of fine-grained locking over a range of application characteristics, including size of shared data, time spent in the critical section, and contention between threads. Our ESTM implementation benefits from the effective use of L1 memory, a feature which is built into our STM model but which cannot be directly utilized by traditional locking mechanisms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming

General Terms Design, Experimentation, Performance

Keywords Embedded Systems, Multicore, Software Transactional Memory (STM), Synchronization, Locking, Transactions

1. Introduction

When general-purpose processors hit the power wall due to increasing clock frequencies, chip designers turned to multiple symmetric cores for increasing performance. In contrast, embedded systems have been utilizing multiple heterogeneous cores for some time [17, 32]. However, since they are typically clocked at lower frequencies, and thus have not hit the same power wall, embedded processor manufacturers have only recently begun to embrace the transition to multicore chips in the form of Symmetric Multiprocessing (SMP). Nevertheless, embedded processors can benefit from adopting a thread-level SMP parallel programming paradigm for the same reasons that general-purpose systems have. Specifically, running embedded applications in parallel can improve performance while keeping the clock frequency the same. If we lower the clock frequency and voltage on each core, we can utilize two cores effectively to achieve lower power, while maintaining a given performance level. Since embedded processors are often used for portable, battery-powered devices, energy reduction provides a powerful motivation for considering a multicore design path.

Unfortunately, parallel programming presents new challenges, such as managing race conditions and concurrent access to shared resources. Traditionally, locks have been used to manage concurrent threads in parallel programs, though this method has severe limitations. Coarse-grained locking is relatively simple to implement, but in the presence of significant contention, can effectively serialize program execution and limit performance. Fine-grained locking can potentially increase parallelism and thus produce performance benefits, but is notoriously difficult to implement and can result in concurrency bugs and deadlock.

Transactional Memory (TM), first proposed by Herlihy and Moss [26], addresses the problems associated with locking by providing the potential for performance on the order of fine-grained locking, with the ease-of-programming and robustness of coarse-grained locking. A *transaction* is a sequence of operations which is executed speculatively. If there are no conflicts with the addresses accessed in the transaction, the memory operation will *commit*, and all its changes will become permanent to shared memory. If there are memory conflicts, the transaction will *abort*, at which point all associated changes are discarded and the transaction will re-execute.

Transactional memory approaches can be broadly classified into one of three types: 1) Hardware Transactional Memory (HTM), 2) Software Transactional Memory (STM), or 3) Hybrid Transactional Memory (HyTM). HTM implementations provide lower overhead than STM implementations, but have architectural limitations. Specifically, transactions must fit in cache and cannot be preempted by the scheduler, and are thus limited by size and execution time. STM implementations, on the other hand, require more overhead to manage the transactions. However, transactions can be any size and run for any duration, and can support nesting. Finally, HyTM implementations incorporate strengths of both methodologies, by using HTM whenever possible and resorting to STM if transactions do not fit the limitations of HTM.

In this paper, we present Embedded Software Transactional Memory (ESTM)—the first full implementation of Software Transactional Memory for an embedded system. As embedded systems are specialized by nature, implementation decisions and optimiza-

LCTES'09, June 19–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-356-3/09/06. . . $5.00

tions which work for one class of applications may not work for others. Our ESTM is targeted toward guidance and navigation embedded systems running operations over large matrices and vectors. We implemented ESTM on the Analog Devices Dual-Core Blackfin with the Visual DSP++ Kernel (VDK).

Embedded STM is a library-based STM written in C, presenting a simple API to embedded systems programmers. Because most embedded systems applications are written in C or assembly, our STM will interface seamlessly with the procedural programming style of embedded applications. In this paper, we discuss key design decisions made in developing our ESTM, as well as the API and its usage. We test our implementation and compare it to traditional locking, both fine-grained and coarse-grained, and show that in many cases, the overhead of the STM is amortized and even overshadowed by the performance increase due to the efficient internal memory usage built into our STM.

2. Related Work

2.1 Hardware and Hybrid Transactional Memory

Hardware transactional memory was first proposed by Herlihy and Moss [26]; their elegant solution added extensions to the multiprocessor cache coherence protocols and a *transactional cache* for use in transactional operations. However, transactions were both spatially and temporally limited, in that they were able to access only a limited number of memory locations and their runtime could not exceed a scheduling quantum as they could not survive a context switch.

This problem led to research in *unbounded* HTMs; unbounded implementations provide mechanisms for transactions to commit even if they exceed a system's architectural resources and/or scheduling quantum [5, 10, 20, 33]. However, this flexibility comes at a cost in hardware complexity, making most implementations impractical for modern processors. Another approach to managing arbitrary transaction sizes is Hybrid Transactional Memory (HyTM); HyTMs operate on the assumption that most transactions are small enough to run within the limits of a bounded HTM, and thus would benefit from the performance of a simple HTM implementation. However, if a transaction overflows the architectural resources available, a STM is provided as a failover solution to manage arbitrarily large or long-running transactions [9, 12].

2.2 Software Transactional Memory

Early predecessors of Software Transactional Memory, like *Multi-word Compare-and-Swap* (MCAS), provided concurrent wait-free or lock-free access to shared objects [6, 7, 19]. Earlier implementations limited parallelism with overly-restrictive comparisons [23]; Anderson and Moir remedied this problem with their MWCAS, which permits parallel operations on disjoint sets of objects, and presents a "helping" mechanism by which one process helps a competing transaction to complete [6]. Anderson et al. later presented additional helping schemes for use on priority-based systems [7].

The problem with MCAS operations is that they do not account for concurrent read-parallelism, i.e., they cannot detect whether a thread accessed memory that was subsequently modified by another thread's MCAS operation. Unless the programmer keeps track of these references, memory may not be consistent through the thread's operation.

2.2.1 Non-Blocking Implementations

Many of the first STM implementations were non-blocking, which requires that a thread that halts indefinitely or fails cannot prevent other threads from making progress. The strongest guarantee of progress is *wait-freedom*, followed by *lock-freedom*; early concurrency mechanisms were lock-free or wait-free [6, 7, 19, 31, 37]. Later implementations relaxed these restrictions and pursued a weaker *obstruction-free* model which results in a less-complex implementation [21, 25, 30].

The first STM model—as well as the terminology "Software Transactional Memory"—was created by Shavit and Touitou [37]. They found that the cooperative helping mechanisms of previous MCAS implementations resulted in extra contention and eliminated the recursive helping structure. They were the first to call their implementation a *static* STM: that is, the data set to be accessed by the transaction, and the transactions themselves, are known in advance.

The next phase of STMs were *dynamic*, in that the memory usage of a transaction did not need to be known in advance. Herlihy et al. claim to be the first to present Dynamic STM (DSTM) [25]. This implementation is known to have performance problems because all data is accessed through a double-level of indirection [14], resulting in cache misses. The authors comment that their work for DSTM was more concerned with the basic transactional model and run-time techniques, and less concerned with providing a simple and flexible API for programmers. They correct this with DSTM2 through *transactional factories*, which allow programmers to "plug in" their own synchronization and recovery mechanisms [24].

Fraser and Harris present three unique APIs to aid in concurrent programming [19]. The first is an MCAS reminiscent of the early MCAS implementations. The second is a *Word-based STM* or *WSTM* which solves the read-parallelism problem of MCAS, but may result in lower performance compared to MCAS. Their last API, *Object-Based STM* or *OSTM*, presents a more straightforward implementation than WSTM and often runs more quickly. It accesses objects through a single level of indirection, which Marathe et al. suggest leads to better performance than DSTM when contention is low or transactions are largely read-only [29].

2.2.2 Blocking Implementations

Ennals' work showed that obstruction-freedom is both unnecessary and interferes with performance-boosting optimizations [16]; as a result, blocking STMs were introduced and grew in popularity [14, 35].

In their work, Saha et al. discuss and evaluate STM tradeoffs, and implement their own STM, called McRT-STM, based on their findings [35]. They show that with certain applications they can approach or beat fine-grained locking with enough processors. Adl-Tabatabai et al. integrate the McRT-STM with an optimizing JIT compiler, and show that their compiler optimizations can reduce the STM overhead and safely eliminate redundant STM operations [1]. Saha et al. also develop architectural support for McRT-STM with instruction set architecture (ISA) extensions and call it Hardware Accelerated STM (HASTM) [36]; they show that it scales as well as STM and better than HTM.

Another interesting approach is called *Transactional Locking* or *TL* [14]; it presents a strong case for blocking STMs and compares lock acquisition at both encounter time and commit time, and finds that commit-time algorithms have the best scalability across typical contention ranges. In follow-on work [13], TL is improved by introducing a global version clock. The new *TL2* eliminates the requirement of a "safe" running environment, so that a thread will not be operating on inconsistent memory states.

2.3 Embedded Transactional Memory

It is worth noting that, though there has been no prior work on embedded *software* transactional memory, Ferri et al. explored a hardware transactional memory solution for MPSoCs [17]. They modeled their implementation after the original TM of Herlihy and Moss [26], using a small, fully-associative *transactional cache*. This transactional cache manages all memory accesses during a transaction and is accessed in parallel with the L1 cache.

3. STM Design Decisions

Given that this is the first implementation of Embedded Software Transactional Memory, we cannot apply past research directly. There has, however, been a considerable amount of work done classifying and evaluating STM design decisions. We can look at the characteristics of these decisions to determine which are best suited to an embedded multicore system. This section discusses these design decisions and motivates the choices made in our implementation.

3.1 Static vs. Dynamic STM

Transactions can be broadly classified into two categories: static and dynamic. *Static* transactions [31, 37] require that data accessed within a transaction, as well as the transaction itself, are defined in advance. In dynamic STMs [1, 19, 24, 25, 30, 35, 36], the set of locations accessed by a transaction is not known in advance, as transactions and transactional objects are created dynamically. This STM is particularly well-suited for dynamic-sized data structures, such as trees and lists [19].

Though fine work has been done in the area of dynamic STM, we believe that the embedded system environment is better suited to the older static STM. A major concern in an embedded environment is keeping the overhead of the implementation low. Though dynamic STM implementations provide greater ease-of-use to the programmers using them, there is a greater overhead in both execution time and implementation complexity. Since embedded systems software developers fine-tune their applications, they already have intimate knowledge of its memory behavior. Since our targeted embedded applications typically have strictly deterministic (i.e., static) data structures, we are not as concerned with complex heap-allocated dynamic-sized data structures. Consequently, dynamic allocation is not currently supported within transactions, though lightweight support (such as specialized `malloc` and `free` operations [12, 35]) will be supported in future implementations.

Though we sacrifice some ease-of-use by limiting the ESTM implementation to static transactions, their use will minimize overhead and complexity. Even so, we maintain that this STM implementation will still require less effort than fine-grained locking.

3.2 Granularity of Conflict Detection

An important consideration in any STM implementation is the granularity of the memory accesses in a transaction. A word-based or cache-line-based scheme detects conflicts across a range of memory locations [11, 19, 35], or, as described by Herlihy et al., works by intercepting direct memory accesses [24]. A word-based STM [11, 19] requires its own metadata to be maintained separate from the data itself, and often code must be inserted around every memory access to indicate whether it is a transactional read or write.

Object-based conflict detection [19, 24, 25, 28, 30, 36] operates at a different granularity. In an implementation done in an object-oriented language like Java [24, 25, 36], conflict detection is done over an abstraction of memory—the object—rather than an address range of memory. Two transactions writing to disjoint elements of the same object may create a conflict even though they are writing to disjoint memory locations. For this reason, object-based implementations may have a higher conflict rate than a word-based approach. However, the implementation is simplified, and it presents a more intuitive interface to the programmer.

Our implementation is closest to a word-based approach, though the granularity of conflict detection is across an entire array, or across a field of a struct (rather than the entire struct), so it is presented intuitively to the programmer as an object-based STM. In a language like C, which lacks the constructs of an object-oriented language, it is difficult to implement a pure object-based STM. Additionally, the ability to manipulate pointers makes C suitable for a word-based approach. Further, through the use of *address books* (described in Section 4.1.1), we do not require that code is inserted around every memory access, as in previous word-based STM implementations.

3.3 Blocking vs. Non-Blocking Implementations

Initially, many STM models were non-blocking—that is, they did not require the use of locks in their implementations [19, 24, 25, 30]. Later research demonstrated the merit of a blocking or lock-based approach [1, 14, 16, 35, 36]. Lock-based approaches are less complex and have less overhead, and research has shown them to be faster than their non-blocking counterparts [14, 16, 35]. The downside of using a locking implementation is the possibility of deadlock. However, locks are only used in the STM code, not in application code, so the programmer does not need to worry about them in his or her application design. Further, deadlock can be eliminated with a timeout mechanism within lock acquisitions.

As embedded applications are generally fine-tuned and optimized to achieve best performance, we do not feel the overhead of a non-blocking implementation is justified. We take steps (as described in Section 3.4) to minimize the effects of the locks. Additionally, with the reduced instruction set of the Blackfin processor, there are no atomic load-link/store-conditional or compare-and-swap instructions, eliminating most obstruction-free implementations from consideration. As the hardware does provide an atomic test-and-set lock instruction, the decision to implement a blocking STM is an easy one.

3.4 Object Acquisition

The acquisition of an object occurs when a transaction asserts ownership of the object in a non-blocking implementation or when a transaction acquires the lock for the object in a blocking implementation [28]. There are two variations for lock acquisition: an *eager acquire* and a *lazy acquire*. In eager acquisition [16, 24, 25, 35], the objects are acquired as memory locations are accessed; in lazy acquisition, the objects are acquired only at commit time [13, 19]. The benefit of an eager acquisition is that conflicts between transactions are detected early, so transactions that will eventually abort do not perform useless work. Using a lazy acquire, a transaction that will eventually abort does not hold the lock for the duration of its execution [30].

We chose the latter methodology to minimize the time the lock is held, and to prevent a lock from being held by an aborting transaction. Locking is performed twice: 1) once very briefly at the start of the transaction, and 2) later at commit time for a longer period of time. We have a single global lock associated with the initialization phase of a transaction, in addition to version numbers associated with individual memory addresses. Our motivation for this global lock is that it allows each transaction to obtain a consistent view of memory at initialization for use throughout its execution.

We also account for a common argument against blocking STMs: that a preempted thread may be holding onto a lock, which may cause another thread to deadlock or waste cycles until the original thread is context-switched back in. We solve this problem by using kernel functions to temporarily prevent the scheduler from preempting a thread while it is holding onto the lock.

3.5 Write-Buffering vs. Undo-Logging

Transactional implementations rely on one of two methods for maintaining consistent views of memory when transactions abort. A *write-buffering* mechanism creates a local copy or buffer of all data used by a transaction; all operations are performed on the local copy, and the data is written back to shared memory only at transaction commit. In an *undo-log* implementation, all writes occur *in-place* to the shared memory location, with a consistent view of memory saved in an undo-log. In the event of a transaction abort, the shared memory reverts back to its previous state using the data from the undo log.

Though Saha et al. show the undo-log implementation has better performance [35], we chose a write-buffering implementation due to the unique embedded system memory architecture. While general-purpose computers have large amounts of cache to reduce slow accesses to main memory, embedded systems often have only a small amount of on-chip memory which can be used as a flat address space or cache (though cache is often not recommended due its non-deterministic behavior and high-power characteristics [8]). To address this issue, there has been a great body of research [15, 18, 27, 34] dedicated to optimizing the use of the *Scratch Pad Memory (SPM)* and minimizing access to slower L2 or external memory.

To keep our STM generic enough to be used on many embedded systems, we assume there is no cache, and provide a mechanism to efficiently utilize the internal memory. The fast SPM is private to each core; therefore, the only shared address space available is in larger-but-slower L2 and external memory, and as a result, shared data cannot be stored in the fastest level of memory. To fix this inefficiency, we use write-buffering, and create a mechanism which works as a software prefetch to bring data to fast internal memory before it is used in execution. Since a write-buffering mechanism requires that each transaction get a local copy of shared data, we always allocate the private copy in the fastest level of memory in which it will fit. This mechanism allows us to utilize the local internal memory more efficiently than if all data remained in L2 memory for all execution, and also fits more data in internal memory than if it were statically allocated there at link-time. Our L1 memory optimization technique is responsible for the performance improvements we see over traditional locking.

3.6 Preventing Starvation

One problem in STMs is ensuring that transactions make progress—that is, ensuring a transaction is not repeatedly aborted due to conflicts with other transactions. Methods for preventing this type of starvation vary in complexity.

One mechanism used to prevent transaction starvation is the *contention manager* [16, 19, 25, 35]. In DSTM [25], a transaction asks the contention manager for permission to abort another transaction. The permission to abort a transaction is given based on a contention management policy.

Despite the prevalence of contention managers in STM implementations, Dice et al. claim that they are unnecessary and can be replaced by a timeout [14]. While we like the simplicity of a timeout, we feel that a more deterministic approach is needed in an environment where there may be real-time deadlines. Since we use commit-time locking, a transaction cannot abort conflicting transactions to commit itself, because by the time the transaction discovers there has been a conflict, the conflicting transaction has already committed. We can, however, provide a contention manager which allows the "losing" transaction to ensure that no conflicting transactions commit until it is able to commit. Our contention manager is discussed in more detail in Section 4.3.3.

4. Embedded Software Transactional Memory (ESTM)

For each transaction, the programmer must declare two objects: 1) a `Transaction` object and 2) a local `AddressBook` object, both described in Section 4.1. They must delimit the transaction with one function call each to start and end the transaction. Finally, they must specify the shared memory addresses which will be accessed within the transaction with one function call per variable or array. The `AddressBook` and `Transaction` data structures are described below, followed by the API and a detailed description of our implementation.

4.1 Data Structures

4.1.1 ESTM AddressBook Structure

In our ESTM, we utilize the concept of an `AddressBook` to simplify pointer management and allow for as little code transformation as possible. The `AddressBook` is a struct defined by the user and contains pointers to all shared data variables and arrays in the application. There is a single *global address book*, which contains the shared memory (permanent) addresses for each variable or array. Each transaction then has a *local address book*, which contains the addresses of all its local copies. Then, the only code transformation required is to reference variables from the local address book struct. Our motivation was to avoid having to insert code around every memory access, which is tedious and makes code more difficult to read. Consider the simple example of a matrix multiply function call:

```
matmult(result, mat1, mat2);
```

STM implementations requiring a transactional wrapper around each memory access [19, 22] would transform that matrix multiply code to:

```
matmult(temp_result,
        txn_read(mat1), txn_read(mat2));
txn_write(result, temp_result);
```

With our address book mechanism, the annotations should be more intuitive and do not add lines of code:

```
matmult(addr_book->result,
        addr_book->mat1, addr_book->mat2);
```

Future work on providing compiler support for embedded STMs would alleviate the need for manual modification of code, just as compiler support for some general-purpose STM implementations has eliminated the manual wrapping of memory accesses in function calls (as required by library-based STMs).

Since the *address book* is declared and filled in by the user, the naming convention is not required (the user will be passing around a pointer to the structure cast as a `void*`).

In addition to readability, a primary motivation for the *address book* implementation is to make all memory accesses fast. Once the initial setup is complete, an STM memory access is as fast as a native memory access (or if the data is allocated to a faster level of memory, even faster). There is no extra bookkeeping involved for a memory access: it is simply referencing the data through the new address.

4.1.2 ESTM Transaction Structure

The `Transaction` struct maintains all necessary information for the transaction; it contains a unique ID for the transaction, a flag indicating whether it is *read-only* or *read/write* (used for commit-time optimization), the maximum acceptable number of aborts before a forced commit (used by the contention manager), and pointers to both the global address book and its local address book. Additionally, it contains bookkeeping information for every memory access which will be made during the course of the transaction. This metadata includes the address of the variable in shared memory (called the transaction's *read-write list*), the address of the private copy, variable read-only status, the size of the variable or array, and the version number at the time the data was copied into local memory.

```
stm_start_setup(Transaction* trans, void** local_address_book, void* global_address_book,
                int address_book_size, int max_aborts);
stm_open_mem(Transaction* trans, void** shared_address, int size_bytes, int read_only);
stm_end_setup(Transaction* trans);
stm_end(Transaction* trans);
```

Figure 1: Embedded Software Transactional Memory API

4.2 Embedded Software Transactional Memory API

There are only four functions in the ESTM API; the first three are called at the start of the transaction, and the fourth is called at the end of the transaction. The API is shown in Figure 1.

4.3 ESTM Implementation

4.3.1 Transaction Setup Phase

Within the body of the `stm_start_setup` function, the member variables of the `Transaction` object are filled in. Space in local memory is allocated for the local address book, and the contents of the global address book are copied to it. Each transaction must have a consistent view of memory for its execution. Therefore, no transaction can be allowed to write its results to shared memory while another transaction is actively copying data from shared memory to its private memory. To maintain a consistent view of memory, we use a `read_counter`: a global lock which keeps track of how many transactions are currently in the setup phase. This `read_counter` is incremented within the call to `stm_start`.

Next, the programmer makes one call to `stm_open_mem` for each shared variable or array which is accessed within the transaction. For each call, a local copy of the data is allocated in the fastest level of memory in which it will fit, and the data from shared memory is copied to the new address. The pointer in the local address book, which previously pointed to the shared data address, is redirected to the new local copy. Thus, for the rest of the transaction, all accesses to that address will be automatically made to the local address. Finally, the attributes of the memory access are saved in the `Transaction` object, including the shared and local memory addresses, the data size, the version number, and read-only status.

The last step of the setup is a call to `stm_end_setup`. At this point, the `read_counter` is decremented so that a transaction that is waiting to commit can do so without disrupting the consistent state of the new transaction. In this function, we also check the status of memory accesses to see if all are read-only; if so, we declare the entire transaction to be read-only allowing for some optimization in the commit phase.

With our setup complete, the program continues with its normal, original execution. The body of the original code is not transformed, except that variables are now accessed through the local address book, as described in Section 4.1.1.

4.3.2 Transaction Commit Phase

At the end of the transaction, a call is made to the function `stm_end`. Within this function, the transaction must first ensure that no other transactions are active in the setup phase, at which point they would be acquiring consistent copies of shared memory. If the `read_counter` is greater than zero, the transaction yields the processor to give a competing thread the opportunity to complete its setup phase. The transaction also acquires the global lock to ensure that no other transaction can enter its setup phase, nor can another transaction attempt to commit its results. The transaction acquires the version numbers for all the memory accesses in its read/write list, and if none of those version numbers differ from the versions recorded in the setup phase, the transaction is allowed to commit.

If the transaction commits, version numbers are incremented for memory addresses in the read/write set, though not for read-only memory accesses. Data is copied from local memory to shared memory using the information stored in the `Transaction` object. The private data is then freed, along with the local address book.

If a memory address in the write list is found to be out of date—that is, its version number has changed since the setup phase—then the transaction must abort. It consults with the Contention Manager (described in Section 4.3.3) to determine whether it can block other transactions from committing in the future to allow itself to commit in its next iteration. It then frees the memory it allocated for local data and gets a fresh local address book. It resets the parameters of the `Transaction` object and then restarts execution from just after where `stm_start` left off.

In this paper we target an embedded system commonly used for Guidance, Navigation and Control Systems, where sensors continually feed updates into the navigation filter. Sometimes, it is acceptable for a thread's read-only data to be an iteration behind the most up-to-date information (we call this "stale data"), as long its view of memory is consistent across all the memory it accesses (it is still not acceptable for *write* memory accesses to be stale, because this would result in interleaving at commit time, violating the atomicity rule of memory consistency). In an optional mode of operation, we allow read-only data to commit regardless of version number changes. This allows for faster commits of read-only data, and automatic commits for read-only transactions. If this policy is not appropriate for the application, a stricter policy can be used by declaring all data as read/write.

4.3.3 Contention Manager

The *Contention Manager* prevents a transaction from starvation. This occurs when one transaction, which in this case we will refer to as the *losing* transaction, accesses the same memory location(s) as another transaction, which we will refer to as the *winning* transaction, and the winning transaction always beats it to the commit phase. In this situation, the losing transaction will continually abort.

In order to prevent thread starvation we propose a contention manager which gives the programmer control over starving transactions: the `max_aborts` member variable of the `Transaction` object. At the start of the transaction, the programmer specifies the `max_aborts` value which indicates how many times a transaction can abort before it blocks commits on conflicting transactions. A `max_aborts` value of 1 will result in the transaction aborting, blocking other transactions from committing, and committing itself on the first retry. Recognizing that a transaction must abort at least once before the contention manager interferes (which may pose a problem for respecting real-time deadlines), we moved the contention manager to the start of the transaction in a more-recent implementation.

When a transaction aborts, it increments a value indicating the number of consecutive aborts. It then asks the contention manager if it is okay to block future commits. If the number of consecutive aborts is equal to the maximum allowable aborts, the contention manager takes over.

While it is too late to abort a conflicting transaction which has already committed, the contention manager can block future transactions from committing if they conflict with the losing transaction. To block future commits, the contention manager stores the addresses of all read/write data accessed by the losing transaction and the unique ID of the losing transaction. The next time any transaction checks to see if it can commit, it checks to see if commits are blocked. If so, it must compare its write set with the blocked write set. If there are conflicts, the transaction cannot commit; if there are no conflicts, the transaction can commit. In the event of two transactions reaching the maximum aborts state at the same time, the later arriving transaction must abort, but can block commits as soon as the first arriving transaction has committed.

5. Experimental Setup

We have implemented our Embedded STM in the C language on the Analog Devices Blackfin Embedded Symmetric Multiprocessor, the BF561 [3]. The BF561 has two cores, each with 100K bytes of private L1 memory, of which 68K bytes can be used for data. The cores share 128K bytes of on-chip L2 SRAM. We used the EZ-Kit Lite evaluation kit for the BF561, which has 16M bytes of off-chip SDRAM.

We used the *Analog Devices VDSP++ Kernel*, or *VDK* [4], to provide support for threading and to manage "unscheduled regions" to prevent preemption of threads while locks are held. We built our STM implementation using the atomic test-and-set lock.

5.1 Embedded Application

Embedded systems are inherently special-purpose, so the optimization techniques and implementations that work for one target application may not work well for another. In this work, we specifically target a *Guidance, Navigation, and Control* (GNC) system. Since the motivator of this work is a GNC algorithm, typical benchmarks used in general-purpose STMs (e.g., operations on linked-lists, hashtables, red/black trees, or counter incrementing) are not appropriate for our platform. We have created a synthetic benchmark to represent a range of characteristics which model a generic GNC system, and provide a thorough evaluation of our ESTM.

The computationally expensive part of a GNC algorithm is the *Kalman Filter*, which recursively estimates the state of a process from noisy measurements, including position, velocity, rotational attributes, and acceleration. The state and error calculations, as well as intermediate data, are stored as matrices and vectors and are calculated largely using common matrix and vector operations. For this reason, we focus on matrix multiply as the core of our synthetic benchmark, and modify parameters of this microbenchmark to represent the range of applications which would utilize it.

The synthetic benchmark consists of a variable number of square matrices. Each thread randomly picks three matrices and multiplies two together, storing the result in the third (it may randomly choose the same matrix multiple times for the same operation). In most cases, this means that two of the memory accesses are read-only, and one is read-write. There are two threads per core and a total of 1000 matrix multiplies performed for each test; threads alternate execution on a core, yielding the processor after every successful matrix multiply operation. In order to evaluate the ESTM's usefulness on our target class of applications, we vary the microbenchmark in three ways:

- Datasize: The size of the shared matrices.
- Time in critical section: The time spent in the critical section is varied by adding additional computations to each thread's execution. These computations operate only on local data, and are thus outside the range of the lock or transaction.
- Contention: The number of shared matrices is varied; since a transaction operates on at most 3 matrices, a larger number of matrices means there is less probability that two transactions running in their critical sections simultaneously will be operating on the same shared matrices.

Though the time spent in a critical section will be large in a GNC algorithm, since the computationally expensive portion occurs on the shared Kalman filter matrices, we vary the percent of time spent in the critical region over a large range to evaluate the efficiency of our ESTM. Given that the navigation filter may operate over just a few states to a few dozen states, we vary the size of the shared matrices between 5x5 and 40x40. Finally, because the time spent operating on the same few matrices may vary depending on the other operations present in the GNC system, we vary the contention for shared data between 5% and 60%.

6. ESTM Experimental Results

In this section, we compare the speedup of our ESTM approach to both fine-grained and coarse-grained locking. While it would be very interesting to compare the performance of our ESTM to general-purpose STMs, this would not be possible because of the limited kernel functionality and hardware support of the Blackfin and would be unfair because general-purpose STMs are more heavy-weight since they need to support a broader class of hardware and applications. The performance results are presented relative to a sequential (single-core) execution. For the coarse-grained lock case, there is a single lock protecting all of shared memory. For the fine-grained lock case, there is a single lock for each matrix (the same conflict granularity as our ESTM); the locks are acquired in increasing order of data address to prevent deadlock.

We also compare these results to the *ESTM-Optimized*, in which we allow transactions operating on stale reads to commit. We vary the parameters of our microbenchmark—data size, contention, and time spent in the critical section—for each of the concurrency mechanisms.

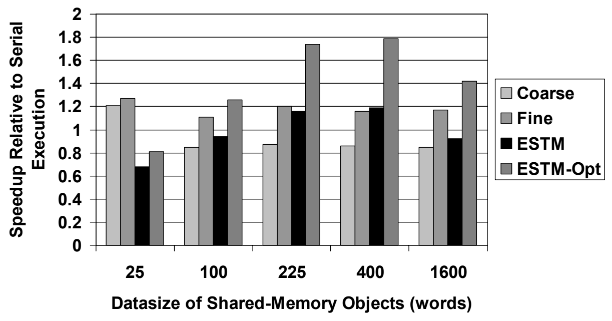
6.1 Varying Size of Shared Data

In the first set of tests, we compare the speedup of the parallel implementations as a function of the size of the shared data. For these tests, we hold the contention rate constant at 30%. Figure 2a shows the effect that data size has on speedup over a sequential implementation for the four concurrency mechanisms.

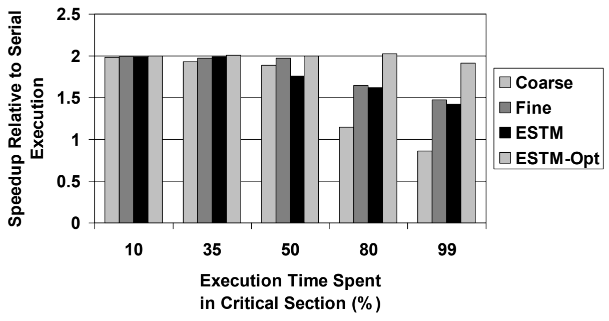
For small shared data sizes, both forms of locking outperform ESTM due to the overhead associated with the ESTM calls and required bookkeeping. As shown in Figure 3a, due to the small matrix size, only a third of the execution time is spent on the critical section. Nearly 45% of the execution time is spent in the STM setup and commit phases. With so little time spent executing original application code, there is no opportunity to offset the overhead of the STM calls.

As Figure 3a shows, more time is spent on the matrix computations relative to the STM calls as the data size increases. The overhead of the STM calls is reduced as the size of shared data (and thus the time spent executing original application code) increases. Coarse-grained locking performs poorly as the execution is essentially serialized even with matrices of only 100 words; this is due to the majority of the execution time being spent in the critical section.

For medium-sized data, ESTM outperforms coarse-grained locking, and approaches the speedup of fine-grained locking. Though there is less overhead in the fine-grained locking implementation, the overhead of the STM calls is amortized by the benefit of operating on data in L1 memory whenever possible, while fine-grained locking must always operate on data stored in slower L2 memory. Figure 3b demonstrates the speedup obtained by running the critical section with ESTM memory management versus the fine-grained execution of the critical section out of L2 memory. Whenever the local copy can fit into L1 internal memory, there is approximately a 1.1x speedup over running the same code out of

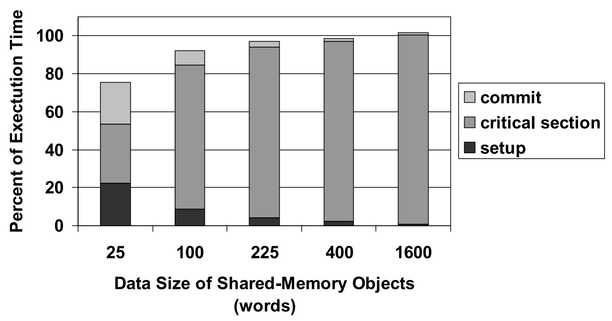


(a) Speedup as a function of size of shared data objects.

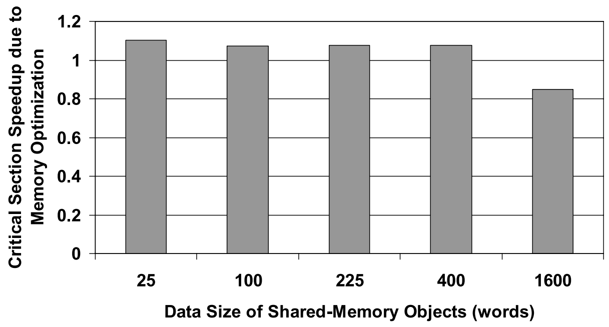


(b) Speedup as a function of percentage of execution time spent in the critical section.

Figure 2: Speedup vs. serial execution for microbenchmark.



(a) Breakdown of execution of critical section and overhead of STM calls.



(b) Speedup of critical section due to ESTM memory management relative to fine-grained execution.

Figure 3: Analysis of application behavior with varying size of shared data.

L2. This is explained by L1 memory accesses occurring in a single cycle but L2 accesses requiring at least seven cycles [2]. If the application were to store shared data in external memory, the speedup obtained with ESTM memory management would be even greater.

At a data size of 1600 words, local copies can not always fit in L1 memory, and occasionally are even copied to external memory. This explains the drop in performance of ESTM relative to fine-grained locking, as seen in both Figure 2a and Figure 3b. Future work will look to mitigate this effect by integrating a smarter memory allocation technique. Currently, our memory allocation scheme naively allocates data to fast memory in the order in which it is encountered; we can improve this by selectively favoring high-profit/low-cost data for placement in L1 memory.

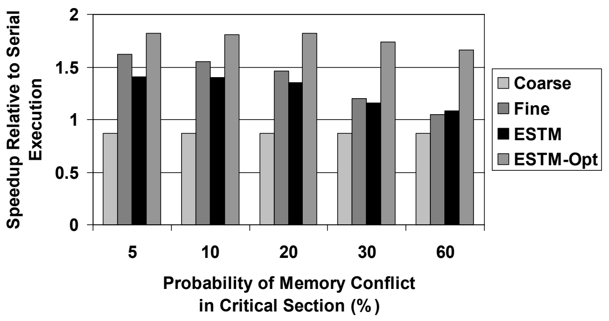
Not surprisingly, ESTM-Optimized outperforms standard ESTM over all data sizes; it also provides better speedup than fine-grained locking in all cases except for operations on small data sizes. This is due to the reduction of conflicts in the ESTM-Optimized. Whereas threads in the fine-grained locking implementation block if any of the three locks it needs are already held, transactions in the ESTM-Optimized implementation will only abort if there is a conflict on the result matrix.

6.2 Varying Time Spent in Critical Sections

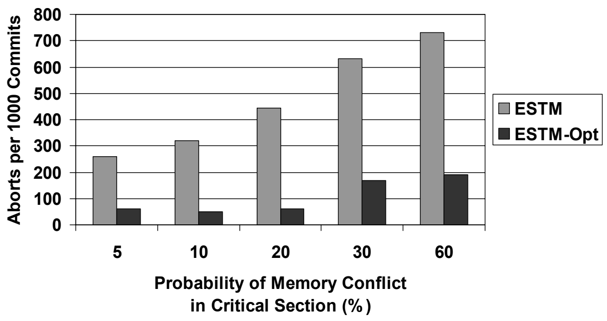
Figure 2b shows how our ESTM compares with both fine-grained and coarse-grained locking when the percentage of execution time spent in the critical section is varied. To remove the effect of the other parameters, we kept the data size constant at 400 words and the contention rate at 20%.

Our results show that, if less than 50% of the time is spent in critical sections, both locking mechanisms and STM result in roughly a 2x speedup over a sequential implementation. This can be attributed to the low likelihood that two threads will be running in their critical sections simultaneously, combined with the 20% probability that two threads running in their critical sections simultaneously will be operating on the same data. The result is that there are few conflicts between threads. This explains why coarse-grained locking performs almost as well as fine-grained locking: there is little opportunity for conflict, so threads will not waste cycles spinning to acquire the single lock. ESTM has the advantage of the optimized L1 memory usage, but with so little time spent within the critical section, this gain is offset by the overhead of the STM calls, and the ESTM speedup is approximately the same as the speedup for both types of locking. From these results, we can infer that, for workloads similar to our synthetic application, applications which are running in critical sections less than 50% of the time can safely use coarse-grained locking, as it is robust and simple to implement, though there is no performance disadvantage to using ESTM in these applications.

As the time spent in the critical section increases, the performance of coarse-grained locking quickly degrades; if nearly all of the execution occurs in a critical section, performance is lower than in a sequential implementation. There is too much conflict be-



(a) Speedup as a function of probability of contention in the critical section.



(b) Transaction aborts per 1000 commits as a function of probability of contention in the critical section.

Figure 4: Speedup and number of aborts for varying contention ranges.

tween threads operating in their critical sections and threads waste time waiting to acquire the single lock. Fine-grained locking and ESTM speedups do not degrade as quickly, as they experience less conflict than coarse-grained. Fine-grained locking and STM perform approximately the same; both are experiencing memory access conflicts, but while the overhead of fine-grained locking is lower, ESTM has the advantage of internal memory optimization. Even with nearly all execution time spent in the critical section, both ESTM and fine-grained locking approach 1.5x speedup over sequential execution. With speedup approximately equal for fine-grained locking and ESTM, we claim that ESTM is a better option for this class of applications across the entire range of time spent in the critical section as it is easier to implement and more robust than fine-grained locking, but with little or no performance loss.

6.3 Varying Shared Memory Contention

Finally, in Figure 4 we look at how contention between threads affects the concurrency mechanisms. For all of these tests we use matrices of size 15x15, where 95% of execution time is spent in the critical section. As expected, the coarse-grained locking is not affected by contention, as the same lock is acquired regardless of whether threads simultaneously access the same or different data sets. Fine-grained locking and ESTM both perform best with less contention, with the benefits of parallelization decreasing as more conflicts occur. As shown in Figure 4b, the rate of aborts increases as the rate of contention increases. This results in code rerunning more often, and explains the overall performance degradation against sequential execution. Since it spins more often waiting for a lock, fine-grained locking performance also degrades, with the result that fine-grained locking and ESTM perform approximately the same across all conflict ranges. Fine-grained locking is more affected by the contention increase than ESTM, and with a 60% conflict rate, ESTM slightly outperforms fine-grained locking. Further, as contention gets higher, a thread using fine-grained locking may starve indefinitely waiting for a lock. The contention manager in our ESTM ensures that no starvation will occur. Because ESTM and fine-grained locking perform approximately the same, it is safe to assume that the easier-to-implement and more robust ESTM is a better option for concurrency control across all contention ranges in this class of applications.

7. Conclusions and Future Work

In this paper, we presented *Embedded Software Transactional Memory* (ESTM), a fully-implemented embedded STM. We discussed six key design decisions to make for any STM implementation, and showed which decisions were best for an embedded platform. We described our ESTM implementation, and showed how two optimizations—L1 memory management and allowing stale reads—could amortize the overhead of the STM calls. We tested our implementation on a synthetic workload representative of Guidance, Navigation, and Control systems. We compared our implementation to both fine-grained locking and coarse-grained locking. In most cases, ESTM produced better performance than coarse-grained locking. Across all ranges of contention and time spent in the critical section, ESTM performs nearly as well or better than fine-grained locking—only performing significantly worse if the size of shared data is very small. With programmer effort and robustness closer to coarse-grained locking, ESTM provides a better alternative to fine-grained locking for our target class of applications.

Given that ESTM is our first implementation of an embedded STM, there is great potential for improvements. The overhead associated with the STM calls proved to be unacceptably high in applications possessing fine-grained shared data. We will look at ways to reduce this overhead through code optimization techniques or in the implementation details in order to support smaller transactions. This in turn will allow us to look at other classes of embedded applications which operate on finer shared data sizes. We plan to investigate the scalability of ESTM by evaluating it on systems with an increasing number of cores. Finally, we will look at the effect of ESTM on power consumption, as reduction of power consumption is an important area of embedded systems research.

8. Acknowledgments

This work has been supported by Charles Stark Draper Laboratories, Inc. Jennifer Mankin was supported on a Draper Fellowship. The authors would like to thank their colleagues at Draper Labs and Northeastern University for their comments and suggestions, and would like to extend particular thanks to Dana Schaa and Micha Moffie for their input and feedback.

References

[1] Adl-Tabatabai, A.-R., Lewis, B. T., Menon, V., Murphy, B. R., Saha, B., and Shpeisman, T. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation* (2006), ACM, pp. 26–37.

[2] Analog Devices, Inc. *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*, revision 1.2 ed. One Technology Way, Norwood, Mass, 02062, February 2007.

[3] Analog Devices, Inc. *ADSP-BF561 Blackfin Processor Hardware Reference*, revision 1.1 ed. One Technology Way, Norwood, Mass, 02062, February 2007.

[4] Analog Devices, Inc. *Visual DSP++ 5.0 Kernel (VDK) User's*

Guide, revision 3.0 ed. One Technology Way, Norwood, Mass, 02062, August 2007.

[5] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (2005), pp. 316–327.

[6] Anderson, J. H., and Moir, M. Universal constructions for multi-object operations. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (1995), ACM, pp. 184–193.

[7] Anderson, J. H., Ramamurthy, S., and Jain, R. Implementing wait-free objects on priority-based systems. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing* (1997), ACM, pp. 229–238.

[8] Avissar, O., Barua, R., and Stewart, D. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems 1* (2002), 6–26.

[9] Baugh, L., Neelakantam, N., and Zilles, C. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture* (2008).

[10] Blundell, C., Devietti, J., Lewis, E. C., and Martin, M. M. K. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (2007).

[11] Crowl, L., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Integrating transactional memory into C++. In *TRANSACT '07: ACM SIGPLAN Workshop on Transactional Computing* (2007), ACM.

[12] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (October 2006), ACM, pp. 336–346.

[13] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing* (September 2006), ACM, pp. 194–208.

[14] Dice, D., and Shavit, N. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (2007), IEEE Computer Society, pp. 21–33.

[15] Egger, B., Kim, C., Jang, C., Nam, Y., Lee, J., and Min, S. L. A dynamic code placement technique for scratchpad memory using postpass optimization. In *CASES* (October 2006), ACM.

[16] Ennals, R. Software transactional memory should not be obstruction-free. Tech. Rep. IRC-TR-06-052, Intel Research Cambridge, January 2006.

[17] Ferri, C., Moreshet, T., Bahar, I. R., Benini, L., and Herlihy, M. A hardware/software framework for supporting transactional memory in a MPSoC environment. *SIGARCH Comput. Archit. News 35*, 1 (2007), 47–54.

[18] Francesco, P., Marchal, P., Atienza, D., Benini, L., Catthoor, F., and Mendias, J. M. An integrated hardware/software approach for run-time scratchpad management. In *DAC '04: Proceedings of the 41st annual conference on Design automation* (2004), ACM.

[19] Fraser, K., and Harris, T. Concurrent programming without locks. *ACM Transactions on Computer Systems 25*, 2 (May 2007).

[20] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st Annual Internation Symposium on Computer Architecture* (2004), IEEE Computer Society.

[21] Harris, T., and Fraser, K. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (2003), ACM, pp. 388–402.

[22] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), ACM, pp. 14–25.

[23] Herlihy, M. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst. 15*, 5 (1993), 745–770.

[24] Herlihy, M., Luchangco, V., and Moir, M. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), ACM, pp. 253–262.

[25] Herlihy, M., Luchangco, V., Moir, M., and III, W. N. S. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing* (July 2003), ACM, pp. 92–101.

[26] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993), ACM, pp. 289–300.

[27] Kumar, T. R., Govindarajan, R., and Kumar, C. R. Optimal code and data layout in embedded systems. In *16th International Conference on VLSI Design (VLSI)* (2003), IEEE.

[28] Marathe, V. J., III, W. N. S., and Scott, M. L. Adaptive software transactional memory. In *DISC 19: In Proceedings of the 19th International Symposium on Distributed Computing* (2005).

[29] Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs in modern software transactional memory systems. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems* (2004), ACM, pp. 1–7.

[30] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., III, M. N. S., and Scott, M. L. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT '06: ACM SIGPLAN Workshop on Transactional Computing* (June 2006), ACM.

[31] Moir, M. Transparant support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms* (1997).

[32] Monchiero, M., Palermo, G., Silvano, C., and Villa, O. Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe* (2006).

[33] Moore, K., Bobba, J., Moravan, M., Hill, M., and Wood, D. LogTM: Log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on* (Feb. 2006).

[34] Panda, P. R., Dutt, N. D., and Nicolau, A. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design Automation and Test Conference* (March 1997), IEEE, pp. 7–11.

[35] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (2006), ACM, pp. 187–197.

[36] Saha, B., Adl-Tabatabai, A.-R., and Jacobson, Q. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), IEEE Computer Society, pp. 185–196.

[37] Shavit, N., and Touitou, D. Software transactional memory. In *PODC '95: Proceedings of the 14th Symposium on the Principles of Distributed Computing* (1995), ACM Press.